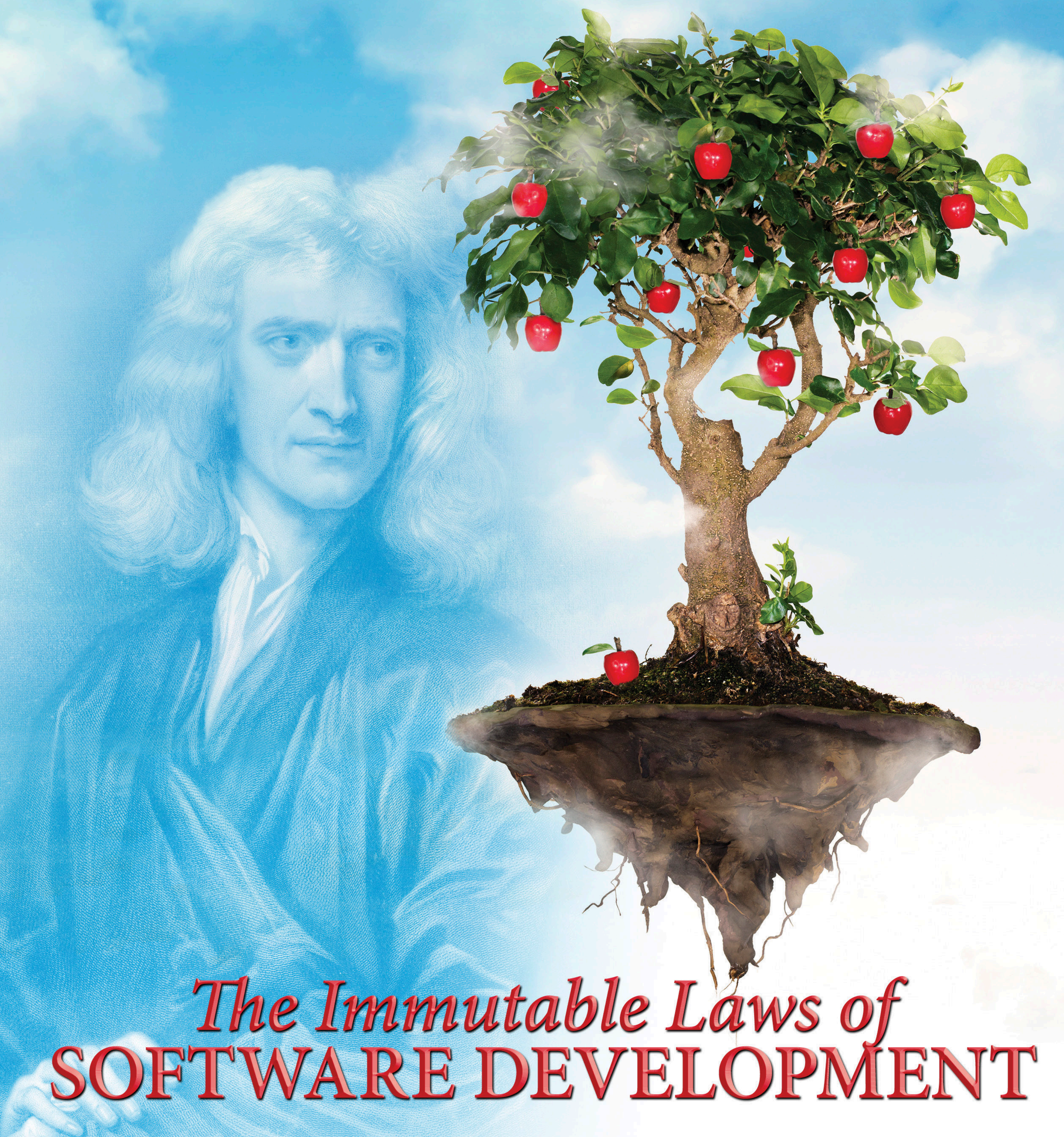
CrossTalk

May / June 2014 **The Journal of Defense Software Engineering** Vol. 27 No. 3



The Immutable Laws of SOFTWARE DEVELOPMENT

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE **JUN 2014**	2. REPORT TYPE	3. DATES COVERED **00-05-2014 to 00-06-2014**

4. TITLE AND SUBTITLE **CrossTalk, The Journal of Defense Software Engineering. Volume 27, Number 3. May/June 2014**	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **517 SMXS/MXDED,6022 Fir Ave,Hill AFB,UT,84056-5820**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

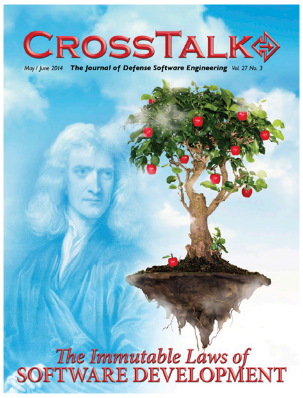
14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**40**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

TABLE OF CONTENTS


CROSSTALK
May / June 2014 The Journal of Defense Software Engineering Vol. 27 No. 3
The Immutable Laws of SOFTWARE DEVELOPMENT

Cover Design by Kent Bingham

Departments

CrossTalk

NAVAIR Jeff Schwalb
DHS Joe Jarzombek
309 SMXG Karl Rogers

Publisher Justin T. Hill
Article Coordinator Heather Giacalone
Managing Director David Erickson
Technical Program Lead Thayne M. Hill
Managing Editor Brandon Ellis
Associate Editor Colin Kelly
Art Director Kevin Kiernan

Phone 801-777-9828
E-mail Crosstalk.Articles@hill.af.mil
CrossTalk Online www.crosstalkonline.org

CrossTalk, The Journal of Defense Software Engineering is co-sponsored by the U.S. Navy (USN); U.S. Air Force (USAF); and the U.S. Department of Homeland Security (DHS). USN co-sponsor: Naval Air Systems Command. USAF co-sponsor: Ogden-ALC 309 SMXG. DHS co-sponsor: Office of Cybersecurity and Communications in the National Protection and Programs Directorate.

The USAF Software Technology Support Center (STSC) is the publisher of CrossTalk providing both editorial oversight and technical review of the journal. CrossTalk's mission is to encourage the engineering development of software to improve the reliability, sustainability, and responsiveness of our warfighting capability.

Subscriptions: Visit <www.crosstalkonline.org/subscribe> to receive an e-mail notification when each new issue is published online or to subscribe to an RSS notification feed.

Article Submissions: We welcome articles of interest to the defense software community. Articles must be approved by the CrossTalk editorial board prior to publication. Please follow the Author Guidelines, available at <www.crosstalkonline.org/submission-guidelines>. CrossTalk does not pay for submissions. Published articles remain the property of the authors and may be submitted to other publications. Security agency releases, clearances, and public affairs office approvals are the sole responsibility of the authors and their organizations.

Reprints: Permission to reprint or post articles must be requested from the author or the copyright holder and coordinated with CrossTalk.

Trademarks and Endorsements: CrossTalk is an authorized publication for members of the DoD. Contents of CrossTalk are not necessarily the official views of, or endorsed by, the U.S. government, the DoD, the co-sponsors, or the STSC. All product names referenced in this issue are trademarks of their companies.

CrossTalk Online Services:
For questions or concerns about crosstalkonline.org web content or functionality contact the CrossTalk webmaster at 801-417-3000 or webmaster@luminpublishing.com.

Back Issues Available: Please phone or e-mail us to see if back issues are available free of charge.

CrossTalk is published six times a year by the U.S. Air Force STSC in concert with Lumin Publishing <luminpublishing.com>. ISSN 2160-1577 (print); ISSN 2160-1593 (online)

CrossTalk would like to thank NAVAIR for sponsoring this issue.

So where does one start when writing about the Immutable Laws of Software Development? As I often do, I went right to my friends at Wikipedia to understand "law" itself and came up with these initial thoughts. First, is it possible, or even desirable, to define law? After all, law is a term that does not have a universally accepted definition. In the broad legal world of international, constitutional, and criminal law, to name a few, it is generally a system of rules and guidelines enforced through social institutions to govern behavior.

When I think about how this definition extends to software, I see the need to transition from philosophically based laws of history (including great thinkers like Plato and Aristotle), to laws where data and observation are combined with documented processes and project roles. A good illustration of this is a phrase usually credited to W. Edwards Deming: "In God we trust; all others must bring data." Capers Jones, just this year, put together a short paper describing many of the laws of software development captured over the last 60 years. In almost all of them there is a reference to large quantities of empirical data from many projects. It is the lasting nature of these laws, in the very fluid world of software development, that lead us to the idea that software laws must be empirical.

As a Team Software Process (TSP) coach I have applied the teachings of Watts Humphrey for nearly 20 years. Much of what Humphrey brought together in the TSP was not revolutionary but rather a gathering of many laws of software engineering from other experts over previous decades. Starting with his experiences and data, I have applied laws such as: the larger a component, the longer it will take to build; project schedules are based on the total estimated hours combined with team members' availability; early defect detection will help schedules remain true and ensure the project will deliver low defect products to the end user. Sources of these software laws come from famous work such as "Quality is Free" by Phil Crosby, "Software Engineering Economics" by Dr. Barry Boehm, and "The Mythical Man-Month" by Fred Brooks.

For these laws to be considered immutable means they are not susceptible to change. While we will continue to go forward with the application of this proven body of work, we must always remain open to change through analysis of data. So keep collecting data, doing postmortem analysis, and evolving these laws as we continue to close this loop.

I welcome you to this issue of CrossTalk and invite you to enjoy and benefit from these great articles.

Jeff Schwalb
NAVAIR Process Resource Team

Collecting Large Biometric Datasets
A Case Study in Applying Software Best Practices

Delores M. Etter, Southern Methodist University
Jennifer Webb, Southern Methodist University
John Howard, Southern Methodist University

Abstract. The lessons and best practices that have become required operating procedure in software development groups can often be applied outside the immediate field of software engineering. This article details a groundbreaking new, multi-year, large-scale biometric dataset that is designed to improve the accuracy and robustness of iris recognition algorithms. We identify several challenges associated with this collection effort and demonstrate how the application of software best practices was able to overcome these obstacles. We believe this list of recommendations represents the current best practices for large scale, long-term biometric collections.

Why Biometrics?

With more than seven billion people now inhabiting our planet, determining an individual's identity has never been more important or more challenging. Biometric algorithms are a form of computer-aided identification that extract and compare various inherent or learned human features. They offer the ability to decipher who someone is, not by what they have, such as an ID card or what they know, such as a password, but by their fundamental intrinsic and behavioral characteristics. Not only are these harder to steal or fake but they also can offer a much lower chance of erroneous identification. For the DoD in particular, which is engaged in international conflicts that can challenge traditional friend-or-foe identification methods, these capabilities are truly transformative.

Iris Biometrics

Iris recognition is a recent technological development that has only become widely utilized in the last decade. First described by Cambridge researchers in the early 1990s, this particular biometric quantizes the intrinsic texture of the human iris in order to automatically determine if two occular images are from the same physical eye [1]. Because individuals with dark or brown irises reflect very little light in the visible spectrum, iris biometric samples are normally collected by sensors that are sensitive to light in the near infrared (NIR) range, which spans from 700 to 900 nm.

Iris recognition algorithms have shown the ability to achieve incredibly low error rates. False match rate (FMR) is the number of times that two different individuals are incorrectly declared to be the same person. False non-match rate (FNMR) is the percentage of trials where a single person appears to not match their own biometric sample, usually requiring the individual to re-submit their test sample. High-quality commercial iris systems can maintain a FMR of one in one million matches while sustaining an FNMR of one in every one thousand attempts [2].

These extremely accurate metrics make iris biometrics one of the few that are appropriate for fully automated population-scale identification programs. Table 1 details some of the large national programs initiated in the last decade. In 2007, the United States military also began utilizing mobile iris biometric technologies. These aptly named devices, known as the Handheld Interagency Identity Detection Equipment (HIIDE) and Secure Electronic Enrollment Kit (SEEK) were deployed to battlefields in both Iraq and Afghanistan to assist with base access, detainee management, local population screening, and special operations missions. By 2009, the Biometrics Identity Management Agency, which executes biometrics initiatives for the DoD, had collected more than 7.5 million iris images in the field [3].

Country	Program Name	Inception	Program Purpose	Estimated Number of Images
India	UID	2009	National ID	1.2 Billion
Indonesia	e-KTP	2012	National ID	170 Million
Mexico	MNID	2010	National ID	100 Million
Middle East (Multiple Countries)	ETS	2004	Immigration Control	50 Million

Table 1 - Population Scale Iris Biometric Programs

Best Practices for Software Development

While software development languages and tools change constantly there are some fundamental principles that have become widely recognized as best practices. At its core, software development encompasses every aspect of product creation. Consequently, best practices in software development can often be seamlessly applied to other technical areas where the goal is the creation of a finished product. This article will demonstrate how four of these concepts, automation, configuration management, documentation and quality control were utilized to address some of the complex problems associated with biometric database construction.

1. A Next Generation Multispectral Iris Biometric Dataset

Motivations

The ability to achieve a FMR of one in every one million matches is truly an impressive statistic. However, the portion of the human population that is enrolled in an iris database is increasing rapidly. Biometric processes must continue to mature so that they can meet this growing demand. This requires development in two key areas:

1. Accuracy – Iris recognition algorithms must continue to demonstrate the ability to reduce false match and non-match error rates in order to support fully automated matching in populations of several million individuals.

2. Robustness – Iris recognition algorithms must continue to sustain performance across increasingly diverse population sets and in increasingly uncontrolled collection conditions.

Recent research has suggested that iris texture changes when illuminated with different wavelengths of light [4], meaning it is possible that several different unique biometric signals can be captured from a single eye (see Figure 1). This discovery has the potential to drive the error rates associated with iris recognition even lower. For example, consider the rare case of two different individuals having matching iris texture in an image captured near 700 nm. By illuminating the two irises with light at some other frequency, it may be feasible to algorithmically determine that the two samples are different, thus avoiding a false-match error.

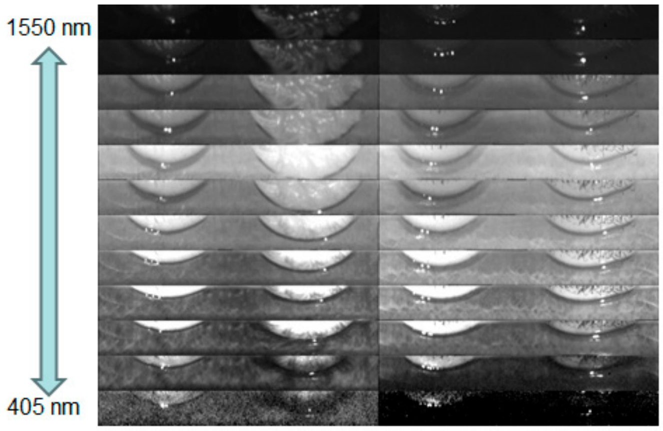


Figure 1 - Unwrapped Iris Texture Illuminated at Different Wavelengths

Approach

In order to stimulate the development of more accurate and robust iris recognition algorithms, a unique data collection was sponsored by the United States government. This collection, known as the Consolidated Multispectral Iris Dataset (CMID), has several notable characteristics that have never been explored in a single biometric collection.

1. Nontraditional Spectrum – Using a custom designed camera assembly (see Figure 2), the CMID captures six images each of the right and left eye across a spectrum that ranges from 400 to 1600 nm. The LEDs used in this experiment have been certified as eye safe by multiple radiation safety experts as well as Institutional Review Boards at both Southern Methodist University (SMU) and the government sponsor. High-resolution visible light images of the ocular region are also taken using a professional photographic camera. Lastly, an image of the left and right iris is acquired using a commercial iris collection device.

2. Duration and Repetition – The CMID collection is in its final (fourth) year with a goal of collecting each subject 16 times over that period.

3. Geographic Separation – The CMID enrolled more than 400 subjects across two geographically separated collection sites in order to increase the diversity of the collected subject pool. Roughly two-thirds of subjects are collected at the SMU research site.

4. Scale – The CMID collects more than 160 iris images per session. The final CMID dataset is expected to contain more than 1 million laboratory quality iris images.

5. Collection of Metadata – In addition to biometric samples, the CMID also captures information about the subjects enrolled in the study such as their gender, eye color, race/ethnicity, and eye health conditions.

6. Manual Segmentation – The first step in all iris recognition algorithms is to use computer vision techniques to separate iris texture from the pupil and sclera. However, these processes may fail on images captured outside the normal 700 to 900 nm spectrum. Consequently, points on the inner and outer iris boundaries are manually identified for each iris image in the CMID.

7. Manual Quality Control – Images in the CMID are also manually categorized into one or more bins based on their quality. These bins denote incidents such as blinks, image blur, and off-axis eye gaze.

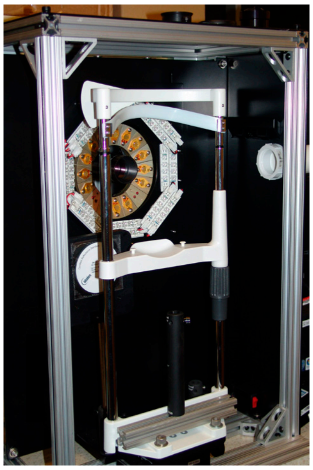


Figure 2 - Consolidated Multispectral Iris Dataset Collection Device

2. Software Best Practices For Iris Database Collection

Executing a first-of-its-kind data collection of this size and with these unique characteristics presented several novel challenges. Without exception these challenges were addressed by applying software development best practices to the biometric data collection methodology. We believe the following represents a list of the current best practices for large-scale multi-year biometric database formulation.

What Can Your Computer Do For You Today?

Automation has long been an enabling technology when developing software. For well-understood tasks, it allows engineers to reduce the possibility of human error throughout the project lifecycle. For example, nightly builds and automated regression testing ensures that this week's code modifications did not break the features added in last week's build. However automation is not synonymous with efficiency. Knowing which tasks to automate and which ones require manual engagement can make the difference between a successful project and one that is underperforming yet over budget.

In a data collection the size of the CMID, automation is a requirement, not simply a desirable feature. Software programs are responsible for nearly everything in the collection process. This includes adjusting the ocular illumination, capturing biometric samples (from all three cameras) and saving the resulting files to the correct location. In order to determine the correct image name, the software must track every variable controlled by the CMID collection (see Table 2). While a small number are entered into a graphical interface by the operator, the majority are ascertained automatically through software processes. Our goal is to prevent a human from ever having to manually save, move, or modify a biometric sample because these operations are prone to error.

Image Specific	Subject Specific
· Collection Site	· Subject Gender
· Source Camera (MS, commercial, photographic)	· Subject Ethnicity
	· Subject Eye Color
· System Version	**Session Specific**
· Subject Identifier	· Contacts Worn
· Left Right or Both Eyes	· Glasses Worn
· Active Wavelength	· Recent Eye Trauma
· Pupil Control State	· Recent Lasik Surgery
· Capture Date	· Recent Other Eye Surgery
· Capture Time	

Table 2 - Consolidated Multispectral Iris Dataset Controlled Variables

One crucial aspect of this effort is the ability to automatically recall the anonymous subject identifier when individuals return for repeat collections. To accomplish this, the iris images captured by the commercial camera are run through a recognition algorithm. The result is used to determine the subject's unique identification number. While it may seem limiting to use an iris recognition system as the identification mechanism when conducting an iris data collection, this function is one of the most crucial steps in any academic biometric capture sequence. Associating the wrong number with a set of biometric images can produce a flurry of inaccurate false match and false non-match errors and call into question the validity of the entire collection.

When performing any biometric collection, system designers should rely heavily on software automation. Especially when tasks are highly repetitive and tedious, every available effort should be made to remove this burden from the human operator. Automated file operations and subject identification is guaranteed to reduce labeling errors across the lifetime of a collection project.

Control The System Configuration Or It Will Control You

Version control and configuration management have long been staples of healthy software development organizations. Software such as Subversion or Git can be used to track changes to a codebase as it matures. When bugs are discovered or misguided development paths realized, these applications allow programmers to revert back to previous stable states.

However, these concepts have rarely been applied to the collection of biometric datasets. Given the longevity of the CMID collection, the geographic separation of the two collection sites, and the deep reliance on automation during the collection process, it was highly likely that software modifications would be required as the project progressed. However, different collection software can inadvertently bias a test, making results appear to degrade or improve when in reality only the capture process has been modified. This presents a classic paradox in test methodology; if on day three of a yearlong test, a process improvement is discovered, do you implement the change at the risk of corrupting the data?

To fully document configuration control within the CMID dataset, a tracking number was integrated into the collection software. This identifier holds the date of the last system modification for a particular site that is then tagged into the name of every image collected over the four-year time span. This allows us to account for any changes in image quality or error rate that might arise from modifications to the collection system configuration.

Monitoring the configuration of the capture setup is crucial for ensuring that inevitable system changes do not bias test results. Each individual biometric sample should be tagged with the configuration tracking mechanism and related documentation provided to end users that details what these numbers mean.

The Most Important Part of the Code, Is Not Code

Documentation can often be viewed as a leading indicator of success in a software project. If the developers cannot use technical documentation to clearly communicate what a group of functions is designed to accomplish, what are the odds it will actually achieve its unuttered objectives? If a project manager cannot concisely communicate, through an end user manual, how to operate a program, can we really assume it works at all?

Meaningful documentation takes on new interpretation when conducting a long-term biometric collection. Previous iris datasets have usually produced academic papers that include voluminous specifications on what was collected but leave out the intricate details of how and why. This is possible when the collection period is relatively short and these details can be maintained in the gray matter of a select few individuals who persist with the project throughout its lifecycle. However, when seeking to maintain high-quality capture standards across thousands of individual collections, conducted by dozens of test operators, at test sites across the country, over an extended time period, the documentation will be the single-most crucial point of failure.

For our collection project, the end-user manual has been the single most modified document in our source tree. It was the first file added to our version control system and is the last file edited before a new software release. It contains detailed, click-by-click instructions on how to use the collection system. It not only tells operators how to setup the hardware and run the software, but why each step is important. It is by far the most accessed and crucial file across the entire project. It is also the hardest to find bugs in, requiring the authors and system designers to continually review the assumptions that each tester will make after reading a given step.

When conducting a long-term biometric test do not discount, save for later, or delegate to the intern, the system documentation. Starting this crucial step early and keeping this document up to date can make the difference between success and failure of the database collection.

If You Don't Care About Quality, You Can Meet Any Requirement

When conducting any long-term, highly involved process it is often easy to forget that all results, especially those arrived at with the help of human involvement, are subject to errors. Quality control is a discipline within software engineering that recognizes this inescapable fact and seeks to identify and mitigate errors in a finished software product.

In what may be a first of its kind effort, the CMID attempted to actively incorporate software quality control principles throughout the collection period. However, instead of only applying these concepts to the finished software product, they were also applied to the deliverables of the CMID; namely the biometric images and the associated metadata.

Three specific quality control measures were taken actively throughout the four-year collection period. The first was to validate that the images being collected by the multispectral capture system would serve their end purpose, namely that they would be appropriate for conducting biometric matches. To satisfy this aim, we actively compared the NIR images collected by the multispectral camera against intra-client samples captured from the commercial iris device. The result of the majority of these operations should be a match. By tracking the rate of non-matches in this subset of images we continually validated that the camera was collecting biometric samples of an appropriate quality.

The second quality control step was also applied to the iris images produced by the collection system. This activity involved identifying the samples that exhibited problematic characteristics, such as blinking, off-axis gaze or motion blur. Tracking these metrics allowed us to actively coach human behaviors on a per-subject basis, which hopefully increases the usability of the dataset. We can also include the categorizations of each image to researchers, allowing them to filter in or out certain classes of imagery, depending on the focus of their analysis.

The final quality control step was designed to validate that the manually chosen points on the inner and outer iris boundaries are accurate representations of these perimeters. As briefly mentioned, every image in the CMID collection is presented to an operator who, with the help of computer software, selects a number of points on the inner and outer iris boundary (see Figure 3). This work is performed by a small team of dedicated staff but is nevertheless very tedious in nature. Consequently, we actively monitor the quality of the segmentations by allowing 1% of the total multispectral imagery to be manually segmented by two or more of the operators. The two different segmentations are compared using an area of overlap metric. By tracking this metric we can not only identify segmentation operators who may need additional training but can also use it to make intelligent estimations as to the overall accuracy of the segmentations across all types of illumination.

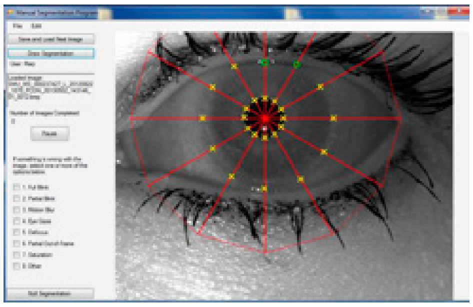


Figure 3 –Manual Segmentation Program.

WANTED

Electrical Engineers and Computer Scientists

Be on the Cutting Edge of Software Development

The Software Maintenance Group at Hill Air Force Base is recruiting **civilians** *(U.S. Citizenship Required).* Benefits include paid vacation, health care plans, matching retirement fund, tuition assistance, and time paid for fitness activities. **Become part of the best and brightest!**

Hill Air Force Base is located close to the Wasatch and Uinta mountains with many recreational opportunities available.

Like facebook

www.facebook.com/309SoftwareMaintenanceGroup

Send resumes to:

309SMXG.SODO@hill.af.mil

or call (801) 775-5555

Actively monitoring the quality of a long-term, large-scale biometric collection is crucial to its eventual success or failure. Simply monitoring raw numbers or gigabytes of data collected, without validating that the samples are well suited for their purpose nearly guarantees disaster. The capture system should be designed around quality control tests (not the other way around) and these tests should produce automated, well-understood metrics that can be tracked by the administrative team. This allows for an understanding of how the test is progressing from a quality standpoint, not simply from a sheer numbers point of view.

3. Conclusions

Software development has a long history of both success and failure. From either case, we learn valuable lessons about the correct way to approach problems, implement solutions and react to the unexpected. It is important to remember that these lessons can often be applied outside the field of software development to assist in other engineering and technical challenges. We have demonstrated how several of these well-established principles have helped resolve some of the complex issues that face research teams when conducting long-term, large-scale biometric collections.

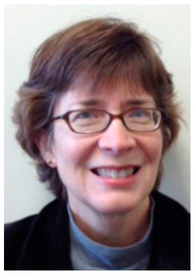
REFERENCES

1. Daugman, John. "Biometric personal identification system based on iris analysis." Patent 5,291,562. 01 March 1994.
2. Daugman, John. "Probing the uniqueness and randomness of iriscodes: Results from 200 billion iris pair comparisons." Proceedings of the IEEE 94.11 (2006): 1927-1935.
3. Quinn, George et al. "IREX IV: Evaluation of Iris Identification Algorithms". NIST Interagency Report 7949 (2013).
4. Boyce, Christopher, et al. "Multispectral iris analysis: A preliminary study51." Computer Vision and Pattern Recognition Workshop, 2006. CVPRW'06. Conference on. IEEE, 2006.

ABOUT THE AUTHORS

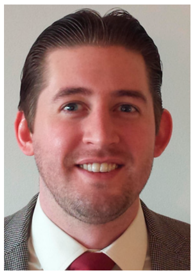
Dr. Delores Etter has been the Texas Instruments Distinguished Professor in Engineering Education and the Executive Director of the Caruth Institute for Engineering Education in the Bobby B. Lyle School of Engineering at SMU since 2008. She previously held academic positions at the U.S Naval Academy, the University of Colorado at Boulder, and the University of New Mexico. She also was the Assistant Secretary of the Navy for Research, Development and Acquisition from 2005 to 2007, and was the Deputy Under Secretary of Defense for Science and Technology from 1998 to 2001. She is also a member of the National Academy of Engineering.

E-mail: DEtter@smu.edu



Dr. Jennifer Webb is a senior researcher in Southern Methodist University's Biometrics Lab, where she has been involved with collection and processing of SMU's Multispectral Iris Image data set for the past four years. Prior to SMU, she worked at Texas Instruments with error-resilient video compression and radar systems analysis. She holds a Ph.D. in Digital Signal Processing from the University of Illinois at Urbana-Champaign and a Master's degree in Computing Science from Texas A&M University.

E-mail: WebbJ@smu.edu



John Howard is currently a Ph.D. candidate in the computer science department at Southern Methodist University. His areas of interest are biometrics, pattern recognition and big data analytics. He also works full time as a research scientist, contracting for various groups in the United States Government. He has extensive knowledge in the areas of computer vision, software development, statistical analysis, and distributed computing.

E-mail: JJHoward@smu.edu

The Problem of Prolific Process

Balancing the Quantity and Quality of Documented Process

Phillip Glen Armour, Corvus International Inc., QSM Inc.

Abstract. What is the optimal amount and level of detail for predefined and documented (and enforced) process for systems development? This question has been debated for decades by software practitioners, computer theorists, and those responsible for resourcing the business.

Introduction

Should we have more process quantity, more process detail, more process options (and more rigorous enforcement of process)? Or should we just leave developers to figure out what they need to do as and when they do it? On one side we have the view that if process is good, then more process must be better—such philosophies can generate enormous volumes of paper-based process documents or their electronic equivalent. On the other hand there are advocates of process so lightweight it hardly exists; with this approach developers are pretty much left to their own devices to work out what to do.

"Big process" assumes that developers will (a) read the immense amount of process documents before or during development (b) understand what is written (c) figure out how to apply it to their situation and (d) make any necessary process adjustments while staying true to the original intent if not the letter of the documented process. This approach also assumes that all this adherence to pre-defined process will make for higher quality systems or make the process faster and less costly or provide a better basis for system compatibility, extension or maintenance. The advocates and authors of such process rarely seem to concern themselves with any negative effects on the morale, creativity, or sense of achievement the developers might experience when they work this way.

On the other hand, those who espouse very lightweight (if any) process assume that developers will (a) actually remember all the activities needed to build a system (b) consistently apply all these steps (c) apply their innate creativity (now liberated by freedom from oppressive process) to more than compensate for anything they miss. These advocates also assume that the developers will have the requisite experience and skill to do all this.

The Second Law

It is clear that the answer lies somewhere in the middle. Predefining everything we should do to build a system is just not possible. If it were, we could automate the process and we would not need people at all. However, allowing each project and each developer to make it up how they work each and every time they build something is a recipe for anarchy. We did that 30 years ago and it did not work very well; in fact the move to big process was fueled in part by the erratic results laissez-faire development gave us. And then the move to Agile was driven by the reaction to the stifling overhead of big process.

It seems that developing process documentation at just the right level is hard. I described this difficulty in the Second Law of Software Process: *We can only define software process at two levels: too vague and too confining [1].*

The irony is intentional and it reflects the dilemma we have when writing process:

- **Too Confining:** if the written process attempts to define all activities under all conditions for all projects building any kind of system, or even a reasonable subset of the same, it becomes very large. Simply because it is very large people will be reluctant to read it. It also becomes difficult to dig through the mountain of documents to find the relevant bit of process just when it is needed. Even more problematic is the constraint that overly large process may enforce. While detailed process is helpful in defining what has occurred before, it cannot explicitly define how to build or test something that is new. In fact, defined process tends to force solutions similar to those that have been built before—specifically the solution scenarios that were used to build the process. It is this inhibiting of the creative process that most lightweight process advocates dislike.
- **Too Vague:** if the written process consists of high-level guidelines, a loose *meta-process* framework within which developers operate freely, ignoring it, modifying it and adjusting it as they wish, the process does not add much value. That is, working with the process and without the process is pretty much the same thing. In this case people complain that the process does not provide useful guidance and direction—the process has no "meat."

Balancing Act

Caught between the hard place of too much documented process and the rock of not enough, how can we find the sweet spot? It is a balancing act. But we also need to take a look at what process is, how we get it, what we expect it to do for us, and how we make sure it works. For an example of how balanced process might be built let us go back to October of 1935.

Failing Fortress

On its second evaluation flight Boeing's Model 299 (the prototype of what would become the B-17 Flying Fortress heavy bomber) crashed. It was flown by Major Ployer Peter Hill who, as one of the Army Air Corp's most experienced test pilots, had flown and evaluated nearly 60 of the Air Corp's newest aircraft. The crash was caused by the pilot's failure to disengage the B-17's gust locks (devices designed to lock control surfaces while the plane was parked). In dealing with the novel and complex demands of preparing and flying an experimental four-engine bomber, Hill forgot a very important step. He just forgot and it cost him his life.

The solution to this kind of problem was not more experience or more training; Major Hill and his co-pilot had plenty of both. The solution was simple process. It was from this beginning that the pilot checklist was born: a simple list of things to do to ensure the plane was set up correctly to fly safely.

Floating Flight 1549

At 3:27 p.m. on January 15, 2009, US Airways flight 1549 struck a flock of Canada Geese at 2,800 feet on its climb out from La Guardia airport in New York City. Immediately after impact, Captain Chesley Sullenberger took the controls while First Officer Jeffery Skiles began working the three-page emergency checklist on how to restart the engines. Four minutes later, Captain Sullenberger landed the unpowered 42-ton aircraft in the Hudson River to the west of 50th Street.

The incredible feat of safely landing a huge airplane on water at around 150 mph received widespread publicity and the pilots and crew were accorded well-deserved accolades. The use of the emergency checklist was not so well known.

Essential Process

The story of flight 1549 gives us clues to what constitutes good process and where process has its limits.

• **Value Added:** given the criticality of the situation, the pilots did not have the latitude to make a mistake in attempting the engine restart. Simply forgetting one step, or working steps in the incorrect order, might have had catastrophic consequences. When stress is high the human brain may not function flawlessly and a simple reminder can help avoid a lot of problems. With their passengers and their own lives at stake, the pilots would not have used *any* process that did not add immediate value.

• **Routine, Well-defined:** the restarting of a jet engine is mostly done the same way each time. There is no value to be added by experimenting with novel ways of powering up a jet turbine and, in this situation, there could have been a lot to lose by using an ineffective process. Process works best for things which are precise, repeatable, well-defined and for which there is no point in doing things differently.

• **Not for "New":** Captain Sullenberger did not use a checklist to actually land the plane in the water; no such checklist exists. Even if a set of rules for landing a large commercial jetliner in a river next to a major metropolis did exist, the crew would not have had time to reference it and land the plane. When something is "new" there are intrinsic limits to what process can achieve.

• **Not if Too Many Specific Conditions:** the pilots had to deal with an enormous amount of information on the wind, the behavior of the plane, communicating with the cabin crew, the passengers and the Air Traffic Control. The combination of these conditions was quite specific to this particular situation. Any "process" would necessarily have to abstract the situation to a set of generalized conditions and the pilots, with only four minutes available to them, would have had to decode these generalizations. Even when there is previous experience available and the situation is not entirely "new," if there are specific conditions that apply to a particular situation, attempting to apply a pre-defined process will take more time and will be considerably less valuable.

• **Succinct:** there are many valuable books on flying airplanes in difficult situations. These pilots did not have time to reference and process them. The engine restart checklist contains only and exactly what is needed to restart an airplane engine under emergency situations.

Process works best when it contains only what is *essential.*

Novel Projects

To some extent, software projects are always "new." We are always building something we have not built before—otherwise we should simply use what we built last time. That said, much of what we do in the business of software *is* repetitive. There are many aspects of our work that can and should be done the same way over and over. But there are also things for which previously defined process does not quite apply at the prescriptive level. Perhaps this is where we can define the boundary of process and extemporization.

What We Know, What We Do Not Know

Building systems consists of two kinds of work: the application of what we already know and the discovery of what we do not know (followed, of course, by its application). By "application" I mean the translation of that knowledge into the executable form we call "software." What we already know, we can call "Zero Order Ignorance"—provably correct knowledge (or its inverse, lack of ignorance).

What we do not (yet) know can be divided into several categories: those things we know we do not know or "First Order Ignorance" (where we have a well-formed question, but do not have the answer) and what we do not know we do not know or "Second Order Ignorance" (where we do not know enough to form even a good contextual question) [2].

Well-defined prescriptive process can work well for Zero Order Ignorance (0OI) and some of First Order Ignorance (1OI), but it cannot work well for the more complex 1OI and for Second Order Ignorance (2OI). Since software projects contain all of these, the process must flex.

Well-defined

Prescriptive process can be developed and should be used for those aspects of systems development which are boring and repetitive and for which there is no value in experimenting or learning a new way of working. A good example of this might be the check-out/check-in of code from a configuration management system. Once a good process has been defined, there is little point in doing it in any other way. Indeed, a lot of bad things might happen if people tried to circumvent the process. These processes always deal with 0OI or the simpler 1OI (for which the well-defined questions typically have a menu-driven answer selection). Here there is value in process.

Innovative

For those aspects of system development that are novel, the process must be intentionally sparse. Developers must be allowed to explore options free from restrictions that might constrain the solution. The developers are dealing with the remainder of their 1OI and also what they might be quite unaware of—their 2OI. Here there is value in explicit lack of process.

Process Transition

As systems development progresses, there can be a natural transition between processes. For example: when we start testing a system, we do not (and cannot) know exactly what to test since to some extent we are looking for things we do not know are not there [3]. Much of the time we are seeking to expose those things we do not know about the system (like what it does do that it should not do). To design tests and test processes, we cannot be highly prescriptive since we do not know what we are looking for. We might have general indications: that tests should focus on predicate boundaries or cover representatives of all (known) input classes, but we cannot say exactly where we will find defects. This process requires opening up the process to the innovative creativity of the testers.

However, once tests have been created, run, and proved, testing can be transitioned to the usually highly prescriptive process we call "regression testing." Setting up an automated regression process before the knowledge is obtained is ineffective and it might force early testing into a high restrictive process mold that constrains testing to the point where it doesn't find what it needs to find.

Write, Test, Measure, Reduce

Good process focuses on the value it delivers. This depends on what has to be done: old or new? Repetitive or innovative? Restart the engines or land in the Hudson? Good process does not over-prescribe where that is not valuable. But there are other aspects of process definition that are often missed:

- **Test the Process:** in many decades of working in software I have rarely seen documented (i.e., on paper) actually tested to see if it works. Paper documented process is often written by people who do not actually use the process they are defining. Even more often these process writers themselves do not use a well-defined, tested and measured process—which is a little ironic. Commercial pilot checklists are written by a team of pilots, aircraft primes, engine manufacturers, and the FAA. They are written by people who use the process. Once the checklists are created they are tested in simulators and in the field to ensure they provide the value that is essential to keeping people safe.
- **Measure the Process:** software process is rarely *measured* to find out if it does, indeed, reduce defects, speed up the process, improve the lot of maintenance staff or any of the other attributes used as rationale for writing, using, and enforcing the process.
- **Reduce:** a further step is necessary and that is to *reduce* the process. As pilot checklists are tested and the effectiveness measured, much effort goes into making them more concise, more pertinent, more valuable, and smaller.

Prolific Process

This intentional and careful reduction of process does not occur in software development—quite the opposite. Once documented process is created, it tends to grow and grow as it attempts to deal with more and more different conditions, to identify more and more different situations, and to cover wider ranges of application. The documented process gets bigger and bigger, more and more complex, requiring more and more effort to read, to understand, and to apply. In doing so it becomes more and more unwieldy and less and less valuable and so less likely to be used at all.

Projects do not crash as spectacularly as the B-17 prototype. But they do crash. To bring them in to a safe landing, we need process that truly supports the business we are in; both the boring repetitive parts and the interesting innovative aspects of what we have to do. The process for each of these aspects should be designed for and support the true nature of the work; such process needs to be more focused and more concise, we should test it and measure it in operation to ensure it is really delivering value.

And we should make it smaller. ◆

ABOUT THE AUTHORS

Phillip Armour is a Senior Consultant at Corvus International and a Principal Consultant at QSM Inc. He has over four decades of experience in software and systems development, was Master Instructor at Motorola University and on the external faculty of two graduate schools. He is the author of The Laws of Software Process (Auerbach 2003) and has penned the column "The Business of Software" at Communications of the ACM since 2000.

Phone: 847-438-1609
E-mail: armour@corvusintl.com

REFERENCES

1. Armour, P.G. The Laws of Software Process CRC Press LLC 2004. p.13
2. Ibid p.8
3. Armour. P.G. "The Unconscious Art of Software Testing" Communications of the ACM. Vol.48 No.1 January 2005

Homeland Security

The Department of Homeland Security, Office of Cybersecurity and Communications (CS&C) is responsible for enhancing the security, resiliency, and reliability of the Nation's cyber and communications infrastructure and actively engages the public and private sectors as well as international partners to prepare for, prevent, and respond to catastrophic incidents that could degrade or overwhelm these strategic assets. CS&C is seeking dynamic individuals to fill critical positions in:

- Cyber Incident Response
- Cyber Risk and Strategic Analysis
- Networks and Systems Engineering
- Computer and Electronic Engineering
- Digital Forensics
- Telecommunications
- Program Management and Analysis
- Vulnerability Detection and Assessment

To learn more about the DHS, Office of Cybersecurity and Communications, go to www.dhs.gov/cybercareers. To apply for a vacant position please go to www.usajobs.gov or visit us at www.DHS.gov.

Acquisition Archetypes

The Hidden Laws of Software-Intensive Development Programs

William E. Novak, SEI
Andrew P. Moore, SEI

Abstract. Many of the behaviors and adverse outcomes that we see in software-intensive programs are the result of "misaligned incentives" between the goals of the individuals involved and those of the larger organization. These interact and play out in recurring dynamics that are familiar to both software developers and managers, but which are still poorly understood. By characterizing the forces within these dynamics explicitly in the form of the "acquisition archetypes" described in this paper we can come to understand the underlying mechanisms that cause these problems, and identify mitigations to help mitigate and prevent them.

Introduction

Software development, especially in the context of defense acquisition programs, displays a set of all-too-familiar outcomes that seem to point to a set of common causes. We see these repeatedly in programs: making up schedule delays by cutting corners on quality activities, postponing risky development tasks until later development spirals, failing to identify critical risks to senior management, underestimating cost by large margins, and many others. We know these patterns and outcomes occur; what we have difficulty understanding is the mechanism which causes them.

The SEI regularly engages with acquisition programs by conducting in-depth Independent Technical Assessments to assess program status, understand the reasons behind specific challenges, and make recommendations for corrective actions and future prevention. These assessments examine different aspects of programs, combining document review and code analysis with face-to-face interviews of program, contractor, and other stakeholder staff. The analyses that have been conducted expose many of the forces that drive these programs, and have provided a detailed portrait of some of the most common pitfalls that programs face.

If we wonder why some of these problems continue to occur, we must realize that it can be difficult to recognize the patterns of the problems that surround us simply because we are standing too close to them. If we do not see the larger patterns that they belong to, we will likely fail to recognize even familiar problems when we encounter them in new circumstances.

This paper explains an approach to thinking about recurring acquisition problems, and presents several examples of "acquisition archetypes" that characterize the structure of the forces that drive various counter-productive software-intensive acquisition program behaviors. These archetypes represent a set of omnipresent, and yet frequently ignored, "laws" of software development. Although ubiquitous, there are ways to get around these laws—and approaches to both mitigating and preventing these behaviors, based on the understanding of the underlying structure, are discussed.

Complex, Dynamic Systems and Acquisition Programs

Our focus in large-scale software development is commonly on the complexity and challenges offered by the system that is being developed. However, one of the reasons that successfully completing a software-intensive acquisition effort can be so hard is that these programs themselves are complex, dynamic systems. They feature complex interactions between the PMO, contractors, subcontractors, test organizations, sustainment organizations, sponsors, and users—all of whom act largely autonomously, and in their own interests. There is limited visibility into actual program progress and status. There are often significant delays between making changes to the system, and seeing their results, making the link between cause and effect within the system unclear. There is feedback that occurs between the decisions and actions of the different stakeholder entities, causing seemingly unpredictable results. The feedback can then produce situations that can escalate despite management's best efforts to control them.

These types of systems can trap people into certain behaviors that are ultimately driven by the system. As a simple example, we can think of the stock market, where people tend to buy when the market is bullish, and sell when it is falling. There is no intention on the part of individuals to cause or contribute to the creation of a market "bubble" or a market crash—and yet that is precisely what our collective behaviors do, even though we are only acting in our own self-interest. Our actions in the context of a complex, dynamic system often have unintended consequences which can make things worse. We are trapped by the ways our rational decisions (as they may appear to us to be) interact with the dynamics of the larger system to which they belong.

Misaligned Incentives and Social Dilemmas

There are incentives within most organizations that work at cross purposes with one another—which are "misaligned"—in that they do not combine to cause actions that produce the desired result. This misalignment can result in ineffective decision-making in which short-term interests take precedent over more strategic longer-term interests, or the objectives of the larger organization can take a back seat to individual or team goals. We may be inclined to think that the recurring behaviors we see in organizations are simply the result of individual personalities and their different styles. While these differences may have significant effects, they cannot explain the recurring nature of these behaviors—and so they are not as important as the contextual structure of laws, regulations, rules, guidelines, and preferences in which people operate. As Peter Senge observed, "When placed in the same system, people, however different, tend to produce similar results." Economists believe that people respond to incentives, and this is correct. We should not expect to rely upon the integrity of people to achieve an organization's goals if the organization's policies and incentives oppose them. While people want to do "the right thing," when they are forced to choose between personal self-interest and organizational goals, the temptation toward self-interest can be too great.

Some examples of misaligned incentives that occur in software-intensive acquisition include:

- The preference for using the most advanced technology, even if it may be immature. The government wishes to provide the most powerful capability to the warfighter, and the contractor prefers to enhance their experience base with the latest technologies—even if the risk of using them is higher.
- The preference for longer duration programs, which allow the government to build greater capability systems, and offer contractors greater staff and revenue stability. However, they increase the risk of scope creep from advancing technology during development.

Many misaligned incentives can be classified as what sociologists and others call "social dilemmas." Social dilemmas describe situations in which the most likely solution to spontaneously emerge is one that may be optimal for the individuals involved, but will likely be suboptimal overall.

One of the most common types of social dilemmas is the social trap. A social trap is a situation where an individual desires a benefit (often by exploiting a shared resource) that will cost everyone else—but if all in the group succumb to that same temptation, then everyone will be worse off, because the common resource will eventually be depleted.

A social trap is often referred to as a "Tragedy of the Commons[1]." The interesting thing about a social trap is that the people involved do not intend to harm themselves or others by their decisions—they are all simply acting in their own self interest—but the "tragic" collective result of depleting the resource is still almost inevitable.

Social traps are not rare—we see examples of them every day: overfishing, traffic congestion, and air and water pollution are all the results of large-scale social traps. These are the unintended consequences (i.e., what economists call "externalities") of our intended activities: catching fish to eat, travelling to other places for work and pleasure, and producing goods and services that we need. In these social dilemma situations, to paraphrase economist Adam Smith, "Individually optimal decisions lead to collectively inferior solutions." Furthermore, because they can appear in so many different forms, they are difficult both to recognize and to fix.

We see an instance of a social trap in joint acquisition programs that attempt to build a single capability that will be used by multiple stakeholders. As more stakeholders agree to participate, they each bring new, unique needs to the joint program office (JPO). If the JPO rejects these additional requirements, they risk driving the stakeholders away, as the stakeholders would generally prefer to build a custom system. However, if the JPO accepts the requirements to satisfy the stakeholders, then doing so will likely drive up the cost, schedule, risk, and complexity of the joint program—and drive the stakeholders away for different reasons.

Systems Thinking

One tool for analyzing complex, dynamic systems is systems thinking—a method that uses the identification of feedback loops to analyze common system structures that either regulate themselves, or may escalate or decline. Systems thinking has its roots in system dynamics work pioneered by Jay W. Forrester at MIT in the 1960s, and views systems as sets of components with complex interrelations occurring between them. A widely used tool for systems thinking is the causal loop diagram, which explicitly represents the feedback loops in the system, showing the driving forces, or causes, of the overall system behavior.

The value of systems thinking is that such diagrams can help to identify the underlying structure of a system, which is what drives the behavior that we see. This is important because without an understanding of that structure, applying solutions to address problems in complex, dynamic systems may have unexpected side-effects that can make things worse. Lasting improvement for such systems may only come from changing the underlying system structure.

One tenet of systems thinking is that the behavior of a system is greater than the aggregate of its individual component behaviors. This "new" system behavior that results, which is generally not an intended result of the system, is called an emergent behavior. Emergent behaviors come about as the result of the interactions among the various rules (physical, legal, social, etc.) that govern the system. Examples of emergent behaviors include the ebb and flow of traffic, the flocking of birds, the meandering courses of rivers, the evolving patterns of cities and suburbs, the synchronized applause of enthusiastic audiences, market "crashes" or sell-offs, and many others. For our purposes here, the unintended consequences seen in systems thinking, both from interacting physical laws, and from the interactions of laws, regulations, policies, guidelines, preferences, and our own decisions and actions, are emergent behaviors.

Software Project Management

Clearly large software development programs are themselves complex, dynamic systems—which may be as complex, or more complex, than the software systems they are developing. Because of their increasing size and complexity, as evidenced by their inconsistent performance and outcomes, our projects may already be growing past the ability of our present management techniques to effectively manage them. While we focus much of our attention on the technical software systems that are our primary goal, we may ignore the fact that software development projects and programs feature autonomous, adaptive elements called "software developers" and "software managers" whose complexity is still poorly understood. The claim has been made by many experts that technical and software engineering issues may not be the primary reasons that many development programs fail. The main culprit in poor program outcomes may be the interactions of the people in the development organizations. Realizing this, as engineers we may look to the technology we understand best as a way to correct and prevent these problems—but as Edward's Law states, "Sociological problems do not always have technological solutions." We may need to look elsewhere to resolve these issues.

In trying to understand how such problems with software development can occur, we need only consider that people who develop systems have incentives to "sell" them with optimistic claims of both substantial benefits and low cost. The prospective customer, who is looking for—and in most cases demanding—a system with significant capabilities at low cost, is often all too willing to believe that this is possible. The combination of incentives amounts to a "conspiracy of hope" among the stakeholders that all will turn out well, when in fact the opposite is more likely.

Program managers will generally focus on maximizing objectives where their performance is measured, and for which they will be rewarded (or penalized if they fail). Ancillary goals that are only desirable for the organization, and do not carry a personal incentive for being met, are unlikely to be achieved. Trying to ensure that a task will be performed by mandating PMs to do it just adds one more thing to an already overloaded plate—it provides only one incentive to do it, leaves in place both competing incentives as well as disincentives for ignoring it, and may not guarantee the quality with which it will be done (especially if the task is viewed as a "check the box" requirement). As an example, planning for software sustainment, while important, rarely has any bonus or penalty attached to it, is not a key performance metric for oversight like earned value management, and the quality of the planning work cannot easily be verified. Thus, the reduction of lifecycle costs through mandated sustainment planning is unlikely to be achieved.

When project managers are rewarded specifically for achieving certain goals, they will likely work hard to make those happen, even if that achievement must occur at the expense of other goals of the organization. There is rarely an incentive for an individual to make sacrifices for the common organizational good—which is, in part, why advancing it is so difficult to achieve.

Acquisition Archetypes

Acquisition archetypes are an adaptation and extension of Peter Senge's system archetypes work. The system archetypes each describe a recurring pattern of dynamic behavior that occurs in complex systems:

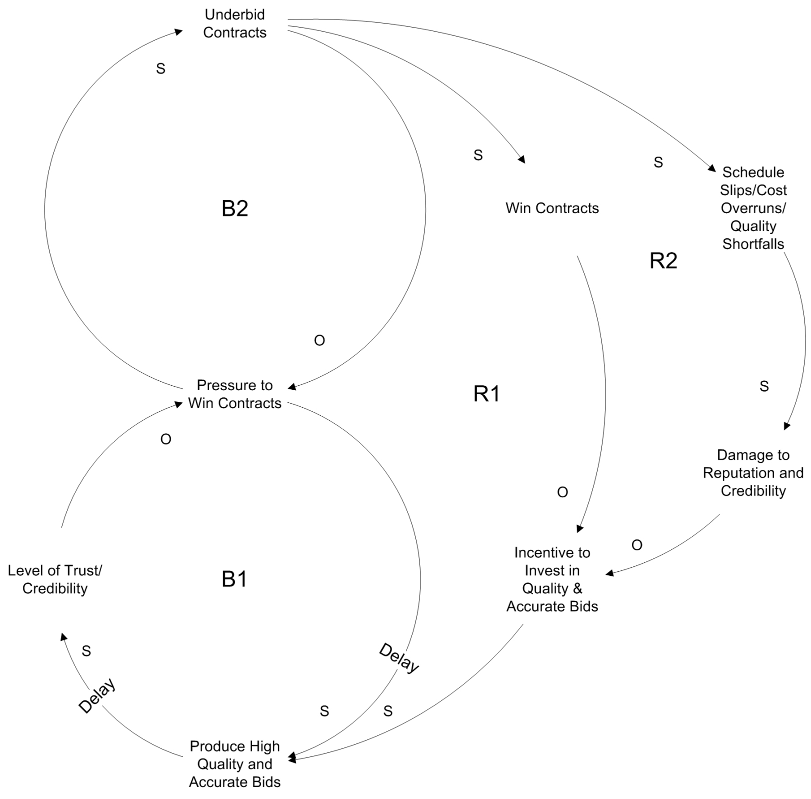
An action appears to be logical and promising—but in practice it has unintended counter-productive consequences to what was desired, or makes other things worse.

The acquisition archetypes adapt the systems thinking approach to describe the recurring patterns of counter-productive behavior in software-intensive acquisition programs. Each of the acquisition archetypes relates the story of a real-world acquisition program that experienced the dynamic, describes how that dynamic occurs on programs more generally, provides a causal loop diagram that can be used to analyze it, and recommends some of the ways the behavior can be mitigated and prevented.

In the following sections three different acquisition archetypes are discussed: Underbidding the Contract, Firefighting, and the Bow Wave Effect. Each one is presented with a summary description of the archetypes, accompanied by a causal loop diagram that depicts the dynamic behavior. A fourth section gives an example of how these archetypes can interact on a program.

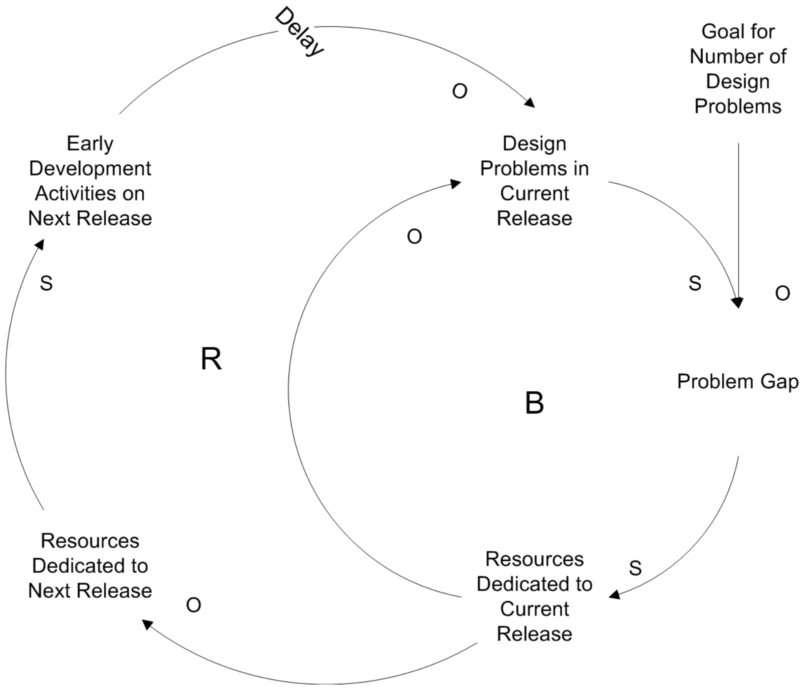
Underbidding the Contract

In the "Underbidding the Contract" archetype shown in Figure 1, the use of the underbidding strategy to win contract awards is successful, and a reinforcing behavior sets in that increases the likelihood of future underbidding. While this approach may have some negative outcomes such as a damaged corporate reputation when the reality of the underbid becomes apparent, thus reducing any remaining intention to produce accurate bids—the advantage of having won the business may be enough to compensate for that. This seeming success is likely to then encourage other contractors to use the strategy themselves to stay competitive, because accurate bids may not be as successful at winning business.



Underbid Contracts
S
B2
S
S
Win Contracts
Schedule Slips/Cost Overruns/ Quality Shortfalls
R2
O
Pressure to Win Contracts
R1
S
O
O
Damage to Reputation and Credibility
O
Level of Trust/ Credibility
B1
Incentive to Invest in Quality & Accurate Bids
S
Delay
Delay
S
S
Produce High Quality and Accurate Bids

Figure 1: Overview of the "Underbidding the Contract" Dynamic[2, 3]



Delay
Goal for Number of Design Problems
O
Early Development Activities on Next Release
Design Problems in Current Release
O
S
S
O
R
Problem Gap
B
Resources Dedicated to Next Release
Resources Dedicated to Current Release
S
O

Figure 2: Overview of the "Firefighting" Dynamic4

Firefighting

In the "Firefighting" archetype that is shown in Figure 2, when a program has a target for the number of allowable defects in the delivered system, and finds itself exceeding that threshold, developers may be shifted from doing early design work on the next release of the system to fixing defects in the current release—which solves that problem. However, unless the total development staff is increased, the lack of designers working on the next release will unavoidably introduce problems into that release. When the next release becomes the current release, it will have even more defects, and the cycle will continue and worsen.

The Bow Wave Effect

The "Bow Wave Effect" archetype shown in Figure 3 shows a pattern of decisions in spiral development which are intended to improve visible progress by postponing riskier tasks in favor of more straightforward tasks that have a higher likelihood of being completed successfully in the near-term. While this approach does improve apparent progress, a backlog of complex tasks that have been deferred to a later spiral is building up like the bow wave in front of a large ship. These tasks will eventually have to be implemented at a time when more of the system has been built, there is less flexibility to accommodate changes, the program may be short on time and budget, and is less able to mitigate the risks those complex tasks may pose.

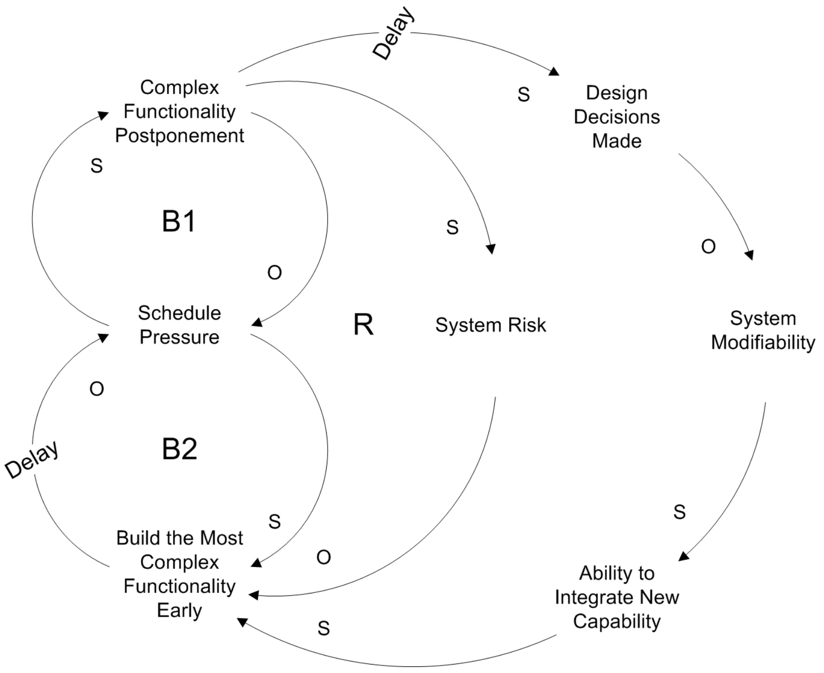

Delay
Complex Functionality Postponement
S
Design Decisions Made
S
B1
S
O
O
Schedule Pressure
R
System Risk
System Modifiability
O
Delay
B2
S
S
Build the Most Complex Functionality Early
O
Ability to Integrate New Capability
S

Figure 3: Overview of the "Bow Wave Effect" Dynamic5

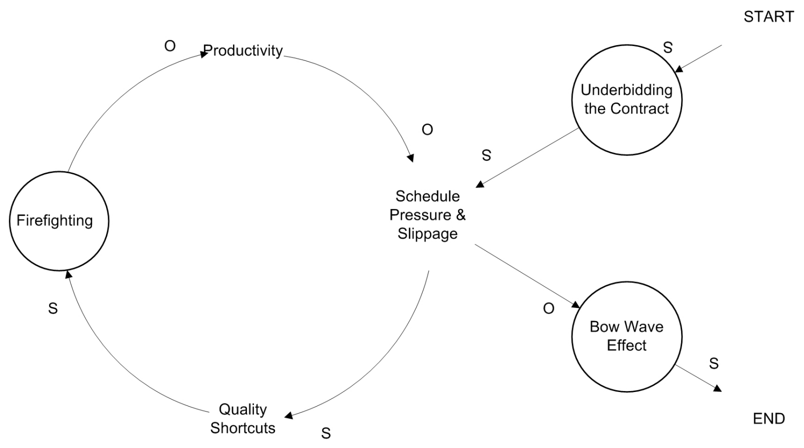


Figure 4: Diagram of Underbidding/Firefighting/Bow Wave Effect Archetypes

Interactions Among Archetypes

Many of the acquisition archetypes are related to one another, and may interact in predictable ways. In most actual programs, multiple interconnected archetypes are seen playing out simultaneously. The diagram in Figure 4 shows one possible set of these interactions.

Initial schedule pressure is created from underestimating effort (underbidding the contract) in order to win the contract. As the schedule pressure increases, a decision is made to delay some of the riskier tasks to be able to show better initial progress to management (the bow wave effect), but in actuality planting a time bomb that the program will trigger late in the development lifecycle when there is no time available to absorb the risk of those tasks. As the schedule starts to slip, certain quality shortcuts begin to occur (missed code reviews, etc.) as a way of reducing the workload and making up time. The increased defects resulting from the weakened quality processes inject new defects into the software—which add to the workload and divert developers from development to bug-fixing (firefighting). With diverted developers, productivity slows, further increasing schedule pressure, and continuing the cycle.

Solving Problems

The primary value of the acquisition archetypes is that they provide a model of the mechanism by which dynamic behaviors occur in systems. Without a model, or with an incorrect model, any proposed solutions to avoid or mitigate the behavior will not address the true root causes, and will be ineffective at best—and disastrous at worst. Lasting improvement will only come from changing the underlying system structure. The causal loop diagrams of the archetypes can be used to make explicit the points at which the dynamic can be influenced so as to improve the typical outcome. With the aid of a causal loop diagram of the situation there are various techniques that can be used to mitigate adverse dynamics. Some of these described by the authors and Daniel Kim are outlined below:

• Reverse the direction of the archetype: It may be possible to turn negative (i.e., adverse) dynamics into positive ones by "running them backwards" and making them beneficial.

• Slow unwanted reinforcing loops: This approach follows the adage, "When you are in a hole, stop digging." While this approach will not eliminate the problem, it will help to minimize the damage, and buy time.

• Accelerate desirable reinforcing loops: The idea here is to make an already beneficial dynamic into one that has even more positive impact.

• Change the value around which a balancing loop stabilizes: In some cases what makes a balancing loop problematic is not its behavior per se, but rather the specific value around which it stabilizes. In such cases we can change the equilibrium value to be something more acceptable.

• Shorten the duration of a time delay: Make it easier to manage the dynamic by bringing the cause and effect closer together in time, to make the linkage between them more evident.

• Find points where a small input change can produce a large effect: Because of a complex system dynamic known as the "Butterfly Effect6," small changes to the inputs of a complex, dynamic system can drive large changes in the outputs. Look for places in the diagram where small interventions can be leveraged by the feedback.

• Identify instances of social dilemmas and apply appropriate candidate solutions: Leverage prior solutions that have been identified.

Applying these techniques to the example archetypes described here provide some practical approaches to breaking out of, or preventing the "Underbidding the Contract" archetype, which include:

- Requiring full technical detail in the Request For Proposal, and thoroughly evaluating proposals
- Investing in, and trusting, a credible government cost estimate
- Establishing a new, realistic cost baseline and replan
- Restructuring the contract
- Looking for tip-offs that underbidding is occurring, such as staff productivity levels that are unrealistic
- Weighting the total technical value of the offer far above bid price in the proposal

Some possibilities for correcting and precluding the firefighting dynamic include:

- Realizing that diverting resources to fix defects only alleviates the symptoms—not the underlying problem—and committing to fix the real problem, with good estimates and more staff
- Revising the plan and/or schedule
- Avoiding investments in new approaches (i.e., improving staff productivity) if the organization is already resource-constrained.
- Doing resource planning with a view across the entire project, rather than locally

Potential ways of mitigating or avoiding the "Bow Wave Effect" archetype include:

- Stopping the use of expedient solutions, but doing so gradually, rather than all at once
- Identifying the root cause for choosing the expedient solution, and changing those incentives
- Considering only options that the organization can realistically handle

Beyond these approaches, the benefit of identifying a problem as an instance of a social dilemma, such as the case of the joint program described previously, is that there is a large set of mitigations and solutions that has been developed to address them.

There are three categories of solutions to social dilemmas:

• **Motivational:** Encourage people to want to change their behavior, because they are concerned about the possible impacts of their actions on others

• **Strategic:** Give people a reason to change their behavior that benefits themselves as well as the larger group

• **Structural:** The most difficult type to implement, the goal is to change the rules of the situation so that people must change their behavior—but this requires some level of authority to implement it, can engender resistance, and may require more expensive compliance enforcement

The motivational and strategic classes of solutions do not require changing the fundamental structure of the situation, and are thus simpler to implement, although potentially less effective than a structural solution.

Motivational solutions, while generally having a lower cost, work best when the participants have little self-interest, which is rarely the case in larger-scale software acquisition programs.

A strategic approach would be to make small changes to the incentive and reward structure of the program, such as improving communications, and making negative behaviors more apparent. While no single such change may significantly mitigate the problem, the aggregate effect of many small changes taken together could have a substantial positive impact. Strategic solutions, however, rely on reputations in longer-term relationships, which are problematic for shorter-tenure active duty servicemen.

The use of a central authority to manage the shared resource (i.e., "commons") at the heart of a social trap is a widespread structural approach, especially in government and military systems where such approaches are already frequently used. However, this approach has unintended side effects such as the incentive it provides to find creative "loopholes" in the mandate (such as a broad interpretation of the definition of "compliance" with the mandate).

There are many other solutions to addressing social dilemmas, such as building trust, exclusion mechanisms, rewarding group achievement (rather than just individuals), and assurance contracts. The choice of the best solution will depend on the specific circumstances surrounding the specific social dilemma.

Conclusions and Future Work

This paper has described a set of example acquisition archetypes that underlie the problems faced by the acquirers and developers of complex software-intensive systems, along with a set of recommended approaches for resolving them. The hope is that this set of acquisition archetypes will be used to help improve acquisition program performance, and that additional research work can be done to produce more acquisition archetypes in the future.

In the future, as the relationships and dynamic effects within programs grow more complex and interact, people will be less able to model the feedback mechanisms of the organizations in their heads to see what the larger emergent effects might be. Fortunately, there are other ways to analyze counter-productive patterns of behavior in programs. The SEI is exploring the development of system dynamics models of software acquisition programs that can simulate the behaviors of such programs. One potential application for decision-making in acquisition programs is the development of interactive educational tools such as management flight simulators to help train acquisition program staff to understand these types of situations better, and thus be better equipped to manage them more effectively.

Another possible application of such a computer model is to answer a question frequently raised by acquisition program leaders: "How will a given change impact the program in terms of cost, schedule, scope, and quality?" It is not feasible to conduct experiments on larger-scale development efforts to answer these kinds of questions. However, a general-purpose, tailorable system dynamics model could help answer such hypothetical "what if?" scenario questions by providing a qualitative analysis of specific program contexts. Such a decision-support tool could improve the quality of key decisions made in acquisition programs—where even small, incremental improvements could provide better program outcomes and substantially improved value and cost savings for the Department of Defense (DoD).

While much more remains to be done to produce better acquisition outcomes, it is hoped that the approaches outlined here can be further developed and applied more broadly to achieve that goal.✥

Disclaimers:

Work for Naval Air Systems Command (NAVAIR) and you'll support our Sailors and Marines by delivering the technologies they need to complete their mission and return home safely. NAVAIR procures, develops, tests and supports Naval aircraft, weapons, and related systems. It's a brain trust comprised of scientists, engineers and business professionals working on the cutting edge of technology.

You don't have to join the military to protect our nation. Become a vital part of NAVAIR, and you'll have a career with endless opportunities. As a civilian employee you'll enjoy more freedom than you thought possible.

Discover more about NAVAIR. Go to **www.navair.navy.mil**.

Equal Opportunity Employer | U.S. Citizenship Required

NOTES

1. The original story of the "Tragedy of the Commons" envisions a group of 19th century herders sharing an area of grazing land called a commons. If one herder decides to graze an extra animal, then that herder receives more benefit from the commons than the others, and at no additional cost to himself. However, if all of the herders follow suit, and add more animals according to the same reasoning, they eventually reach the point where the grass is eaten faster than it can grow, the cattle begin to starve, and ultimately all of the herders lose their livelihood.
2. Causal loop diagrams show how system variables (nodes) influence one another (arrows). The effects of the arrows are labeled "S" for "Same" when both variables change in the same direction, or "O" for "Opposite" when the variables change in opposite directions. Loops formed by the arrows are labeled either "B" for "Balancing" when they converge toward a value, or "R" for "Reinforcing" when they continually increase or decrease. The term "Delay" on an arrow indicates an actual time delay.
3. This diagram is based on the "Shifting the Burden" systems archetype described in (Senge, 1990).
4. This diagram is the "Firefighting" dynamic described in (Repenning, Goncalves, & Black, 2001).
5. This diagram is based on the "Shifting the Burden" systems archetype described in (Senge, 1990).
6. The "Butterfly Effect" refers to the sensitivity of the outputs of a deterministic, nonlinear system to very small changes in the inputs. It was named by the mathematicion and meteorologist Edward Lorenz, and refers to the theoretical possibility of a hurricane forming as the result of a butterfly flapping its wings.

ABOUT THE AUTHORS

William E. Novak is a Senior Member of the Technical Staff at the Carnegie Mellon University Software Engineering Institute, with over thirty years of experience with government software systems acquisition and real-time embedded software. Mr. Novak held positions with GE Corporate Research and Development, GE Aerospace, Texas Instruments, and Tartan Laboratories. Mr. Novak received his M.S. in Computer Engineering from Rensselaer Polytechnic Institute, and B.S. in Computer Science from the University of Illinois at Urbana-Champaign.

E-mail: wen@sei.cmu.edu
Phone: 412-268-7700

Andrew P. Moore is a Senior Member of the Technical Staff at Carnegie Mellon University's Software Engineering Institute with more than 25 years of experience in mission-critical systems modeling and analysis. He has worked for the Naval Research Laboratory and has published widely, including a book on insider cybersecurity threats. Andy received a MA in Computer Science (Duke University), a BA in Mathematics (College of Wooster), and a Graduate Certificate in System Dynamics (Worcester Polytechnic Institute).

E-mail: apm@sei.cmu.edu
Phone: 412-268-7700

Software Engineering Institute
4500 Fifth Avenue
Pittsburgh, PA 15213
(412) 268-7700

REFERENCES

1. Cross, John G. and Melvin J. Guyer. Social Traps. Ann Arbor: University of Michigan Press, 1980.
2. Firefighting. Dir. William E. Novak. Software Engineering Institute. 2012. Animated Short. <http://www.sei.cmu.edu/acquisition/research/archetypes.cfm>.
3. Forrester, Jay W. Principles of Systems. Pegasus Communications, 1971.
4. Hardin, Garrett. "Tragedy of the Commons." Science 162 (1968): 1243-1248.
5. Kadish, Ronald. "Defense Acquisition Performance Assessment Report - Assessment Panel of the Defense Acquisition Performance Assesment Project." 2006.
6. Kim, Daniel H. System Archetypes: Diagnosing Systemic Issues and Designing High-Leverage Interventions. Vols. I, II, & III. Pegasus Communications, Inc., 1993. 3 vols.
7. Kollock, Peter. "Social Dilemmas: The Anatomy of Cooperation." Annual Review of Sociology 24 (1998): 183-214.
8. Madachy, Raymond J. Software Process Dynamics. Wiley-IEEE Press, 2008.
9. Meadows, Donella. Thinking in Systems: A Primer. White River Junction, VT: Chelsea Green Publishing, 2008.
10. Moore, Andrew P. and William E. Novak. "Modeling the Evolution of a Science Project in Software-Reliant System Acquisition Programs." Conference of the Systems Dynamics Society. Boston, MA, 2013. <http://www.systemdynamics.org/conferences/2013/proceed/papers/P1029.pdf>.
11. Moore, Andrew P. and William E. Novak. "The Joint Program Dilemma; Analyzing the Pervasive Role that Social Dilemmas Play in Undermining Acquisition Success." Proceedings of the 10th Annual Naval Postgraduate School Acquisition Research Symposium. Monterey, CA, 2013. <o http://www.acquisitionresearch.net/files/FY2013/NPS-AM-13-C10P01R07-036.pdf>.
12. Novak, William E. and Harry L. Levinson. "The Effects of Incentives in Acquisition Competition on Program Outcomes." Proceedings of the Defense Acquisition University Acquisition Research Symposium. Ft. Belvoir, VA, 2012. <http://www.sei.cmu.edu/library/abstracts/reports/12tr001.cfm>.
13. Novak, William E. and Linda Levine. Success in Acquisition: Using Archetypes to Beat the Odds. Technical Report. Software Engineering Institute. Pittsburgh, PA, 2010. <http://www.sei.cmu.edu/library/abstracts/reports/10tr016.cfm>.
14. Novak, William E., Andrew P. Moore and Christopher Alberts. The Evolution of a Science Project: A Preliminary System Dynamics Model of a Recurring Software-Reliant Acquisition Behavior. SEI Technical Report . Carnegie Mellon University. Pittsburgh: Software Engineering Institute, 2012.
15. Repenning, Nelson P., Paulo Goncalves and Laura J. Black. "Past the Tipping Point: The Persistence of Firefighting in Product Development." Californial Management Review (2001).
16. Senge, Peter M. The Fifth Discipline. Doubleday/Currency, 1990.

The Bow Wave Effect. Dir. William E. Novak. Software Engineering Institute. 2013. Animated Short. <http://www.sei.cmu.edu/acquisition/research/archetypes.cfm>.

Combating the Inevitable Aging of Software Developers

Robert Ball, Stephen F. Austin State University
David Cook, Stephen F. Austin State University
Michael Pickard, Stephen F. Austin State University

Abstract. One of the immutable laws of software evolution is that the developers, along with the software, require sustainment. New college grads are typically drawn to newer technologies and innovative mobile applications. The DoD has software applications that have lifecycles measured in decades, rather than months. The DoD has skilled developers and program managers who have years of valuable experience in the development and sustainment of these long-lived software programs—and these developers and managers are a valuable commodity that cannot easily be replaced. With age comes wisdom, but also, with age comes inevitable decreases in some skills. This article will summarize the effects of aging on computer use, and discuss the proactive steps that can be taken to combat these negatives effects and prevent a decrease in the effectiveness of computer usage skills due to age.

The Inevitable Aging Process

Since the dawn of human civilization man has been seeking the fountain of youth. This insatiable desire to avoid aging has not lessened in modern times. A quick look at plastic surgery trends discussed in three sources gives us a general idea of the desire to look better and younger. Note that the trends are for 2010, the most recent year of released statistics and refer only to the United States[1]:

- Approximately $10.1 billion was spent on plastic surgery in the year.
- There was a 77% increase in procedures from 2000 to 2010.
- Plastic surgery procedure demands increased almost 9% from 2009.
- Approximately 13.1 million cosmetic procedures were performed in 2010.
- Of those procedures, 48% were performed on individuals between 40-54 years old and 25% on individuals 55 and older.

Clearly, people do not want to look older and for good reason. Looking older makes people treat you differently. Pat Moore, a renowned industrial designer and gerontologist spent approximately three years disguised as an 85-year-old woman. When she started her experiment she was only 26 years old.

Pat Moore learned from a professional makeup designer how to create the impression that she was an old woman. Besides extensive makeup she went to such lengths to act old that she taped her fingers to better imitate arthritis and added restraining devices to her back, hips, and legs to better imitate an old woman.

In the end she found that simply looking older makes a dramatic difference in how people treat you. She visited 116 cities in 14 states and two Canadian provinces. She found that with a few, subtle, subcultural exceptions, older people are universally more ignored, thought more incompetent, and less able to perform[2].

There are additional reasons for not wanting to appear older. While there are laws in place to prevent discrimination on the basis of age, there are subtle actions that can result in older workers being forced out of the workplace. For example, older workers generally will command higher salaries as a result of their greater experience; as a result, many hiring managers are inclined to bypass these candidates because of budget considerations. Additionally, there is a growing perception that older workers represent a bigger risk to companies in lost productivity due to medical problems and associated sick days.

Another phenomenon we are currently experiencing is a growing divide in the demographics of the workplace. We now have four generations in the workplace; the Millennials, born between 1980 and 2000; the Gen X'ers, born between 1960 and 1980; the Boomers, born between 1943 and 1960; and the Traditionalists, born between 1922 and 1943. The Millennials have grown up with electronic devices and expect instant gratification - they are very focused on technology. The X'ers are technologically literate, but are very jaded, having grown up with Watergate, the energy crisis, and Desert Storm. The Boomers are very team oriented, but are also driven by a high need for personal gratification. Finally, the Traditionalists are marked by dedication, sacrifice, and a "duty before pleasure" attitude.

These differences may create situations in which generational interactions and acceptance of new technologies in the workplace could be difficult, possibly resulting in confrontations. For example, Traditionalists and Boomers tend not to question authority, but the X'ers and Millennials have been taught to speak up and question authority. Indeed, the two younger generations tend to value recent contributions (what have you done for me lately?) and expect instant feedback, while the older generations value historical contributions, and accept annual (or no) feedback as the norm (no news is good news). These differences can also show up when workers interact with technology, as the Boomers and Traditionalists can be highly resistant toward accepting changes in the form of new technology[3]. While the Millennials and X'ers have different life experiences and communicate with people differently than the Boomers and Traditionalists, there is potential for synergism if they can find ways to exploit those differences.

There are also certain physical and psychological things that happen to us as we age. As we age, there is progressive denaturation of the lens proteins, and the lens becomes thicker and less elastic over time that produces a medical condition called "presbyopia." The result of those changes in the lens is the loss of the ability to change its refractive power, so we cannot change our focus from near to long distance. The refractive power of the lens gradually decreases from about 14 diopters in children to less than two diopters by the time we are 50, and essentially zero diopters by age 70 where the eye becomes fixed focus. This denaturation also affects the optical clarity of the lens, reducing the amount of light transmitted to the retina and distorting color perception as well[4].

The Benefits of Mature Developers

Since our physical eyesight degrades as we age, we would expect that younger adults would be able to read and comprehend what they read faster. However, luckily this is NOT the case. While it is true that older adults do not read at the same speed as younger adults, what is also true is that older adults usually read (and comprehend) faster[5]! Based on crystallized intelligence, people read faster the older they get, as long as they continue to read throughout their lifetime.

Crystallized intelligence is the ability to use skills, knowledge, and experience and is related to verbal ability and the ability to come up with strategies to complete tasks[6]. As long as a person continues reading throughout his life (so that reading skills do not degrade simply due to lack of practice), their reading comprehension and speed also improves. Because of this, older adults read faster in general than younger adults.

Fluid intelligence is the ability to deal with new situations independent of acquired knowledge. Although both types of intelligence increase during childhood and the teenage years, fluid intelligence begins to decline between the ages of 30 and 40 (for most people). However, crystallized intelligence continues to grow throughout adulthood and begins to decline only very late in life.

In other words, an older person may not be able to learn how to do something new as quickly as younger people because of the youth-related advantage in fluid intelligence, but an older person generally can perform a familiar task better and faster than younger adults because of crystallized intelligence.

Do older adults read faster than younger adults on a computer? It turns out that if the font size of the computer is sufficient for the older person's eyesight, then, yes, older adults do read faster than younger adults from computers. In addition, what most people do not realize is that reading from computers is not slower than reading from paper these days. With today's crisp displays, reading from paper and from computers no longer provides a statistical difference in performance. While reading speed may not be statistically different, there is often a preference among older workers to read from paper instead of computer screens. The degree of preference is related to the amount of experience with reading from computers versus paper[7].

There are also a number of other benefits from using larger displays. Using larger displays allows you to see more of the data you are analyzing. Larger displays that show more data at once have been shown to allow people to understand the data faster and to a greater degree of comprehension[8].

Coupling that research with the greater experience and wisdom—crystallized intelligence—that comes with years of working in industry produces a synergistic effect when you can see more data at once. Being able to see more data at once enhances the older person's advantage over youth.

In addition, having a greater view of the data allows one to see and comprehend the data in new and innovative ways. A research study was performed in which expert video gamers were asked to play the same strategic game on different sized displays. They found that the larger the display, the better the strategy the gamers were able to employ and the more they won[9]. Is not "winning" at business often no more than simply understanding the business data and coming up with better strategies than other businesses?

Combating Age-related Skill Deterioration

There are always technological innovations to help productivity. The problem is often that there are too many new technologies to evaluate. A key point to remember about new technologies is that there is a company behind every product. In addition, there is usually a marketing team that works for that company that wants to sell you the technology. The company wants you to think that you have to buy the technology; they want you to think that you cannot solve your problems without it.

There are several extremes that people tend to follow in regard to technology. The first type we call The Hammer. The Hammer is the person who is content to use a familiar technology rather than learn another which might be better.

A famous quote often called the law of the instruments is "If all you have is a hammer, everything looks like a nail."[10] Obviously one technology will not fit all needs, but this type of person tries to accomplish all business tasks with the one piece of technology he already knows.

A variant of the type of person that does not want to accept new technology is The Self-Fulfilled Prophesier. The Self-Fulfilled Prophesier believes that before they have seen or used the new technology—regardless of what it might be—that they will not be able to learn to use it. This person subconsciously and consciously acts in ways that cause him to fail. They fail in learning to use the technology and it reinforces their negative view that they cannot learn new technologies. According to psychology experts, this self-fulfilling failure often actually makes the person happy that he failed[11].

The other extreme is The Marketer. The Marketers embrace all new technology simply because it is new. In our experience they tend to follow one technology company more than others. They absolutely must have any new technology that the particular company introduces.

The Marketer always has the newest, fastest technology, and will tell anyone that will listen why it is the best and why they should buy it, too. In effect, they become an unpaid part of the marketing team of that technology company.

The key to using technologies (both old and new) is to view them as tools for accomplishing a particular task. New technologies come out constantly. If the technology is not useful in helping one accomplish a task, then it is simply a toy to be played with—but not useful technology. On the other hand, if a new technology can be used to help you accomplish a particular task, then the new technology becomes a useful tool.

It is not necessary for a person to learn to use every idea that comes from technology companies. Some of these "new technologies" turn out to be nothing but a toy. On the other hand, it is not wise to fear or ignore new technology. Some "new technologies", when examined, become useful tools. It is also worth noting that these experiences are different for different people. Some individuals will examine a new technology, and discard it as worthless—it is only a vaguely interesting toy. Others, however, will find the new technology interesting and useful—a tool that will multiple their productivity.

For these reasons, neither the Hammer, the Prophesier nor the Extremist viewpoint is correct. Learn the tools that you find useful to help fulfill your tasks and ignore all the toys that accumulate around you. Of course, also be flexible, so that if one of those toys turns out to have potential value, then you would be willing to learn how to use them. The following are "tried and true" technologies that can increase productivity and combat any age-related decreases in certain skill areas.

Physical Adaptations: Monitors

As explained above, our eyes change as we age. Vision declines with age in five dimensions: visual processing speed, light sensitivity, dynamic vision, near vision and visual search[12].

Increasing the size and quality of the monitor can alleviate many of these declines. An aging 17-inch CRT monitor is no match for a crisp, clear, bright 40-inch LCD monitor. Why stop at 40 inches? Why not move up to a 90-inch LCD monitor?

There are several reasons that bigger is not always better. First, the cost of a 90-inch monitor approaches $5,000 or more. A 40-inch monitor can easily be bought for less than $500.

Also, the size of the work area necessary for a 90-inch monitor is not usually feasible due to the second reason—optimal viewing distance. The recommended minimal viewing distance for a 90-inch monitor is more than 8 feet! Indeed, a 40-inch monitor has a minimal distance of 3 to 4 feet, depending upon the light source. A reasonable 30-inch monitor, however, costs less than $250, has a minimum viewing distance of 2–3 feet, and requires little more room than the bulky 17-inch CRT.

Pixel density determines optimal viewing distance. Most contemporary 90-inch monitors have approximately the same number of pixels as a much smaller monitor, thus the larger monitor shows the same amount of data, but the data is just shown physically larger.

Another advantage of feature-rich newer LCD monitors includes increased clarity and brightness of the display. Increased brightness translates into small pupil size, providing increased "depth of field" for aging viewers. This is why older persons typically need a brighter reading environment than younger people—it gives them increased clarity. In addition, the non-interlaced LCD display provides a higher resolution (discussed below), helpful for watching videos or browsing the Internet without the "flickering" that was part of the CRT-era viewing experience.

Often, a more economical solution is to use multiple, smaller monitors. Numerous studies have shown that use of more than one monitor can drastically increase the productivity of people of all ages. Studies confirm a clear pattern of improved information processing. Using multiple monitors allows users to significantly increase the amount of information they can process. Results show that multiple monitors increase comprehension, and that this increased comprehension leads to increased task performance. Recent studies support the increased utilization of multiple monitors[13].

A suggestion on how to leverage the effectiveness of multiple monitors is to use dedicated monitors for increased productivity[14]. For example, email could always be on one monitor, and word processing would be accomplished on another. A popular approach is to use a monitor that rotates for a document view (e.g. a traditional-sized monitor rotated 90 degrees, to resemble the size of a typical page of a document) and another monitor, aligned the normal way, for email and other tasks. It is worth noting that a rotating 27-inch monitor is currently less than $200 and has an optimum viewing distance of less than three feet. For less than $500 a dual monitor setup of very high quality can be obtained.

The above studies suggest that the next time you upgrade your computer system you may want to pay more on upgrading your monitor(s) than your computer speed. When it comes down to total task performance time, larger monitors can help you accomplish your goals faster than a faster computer[15].

Physical Adaptations: Increasing Readability

If you have a hard time reading from computer monitors, there are a number of changes that you can make to your environment to improve the situation. One option is to increase the size of the text and icons on the display. The three most popular operating systems (Windows, Macintosh, and Linux) all permit the user to increase text and icon size.

In addition, you can also lower the screen resolution, which also increases the size of what is shown. Lowering the resolution limits the amount of data that can be displayed at a time, but it always increases the size of all the data for easier viewing.

The aging user should also experiment with display brightness and contrast to find the optimum setting that makes viewing comfortable and effective. Note that on computers with multiple monitors, each monitor can be set to a different brightness and contrast, permitting one screen to be used for videos (lower contrast) and one for document and email (higher contrast). Some users might find that reducing color saturation (moving to black-and-white or grey-scale) might be the optimum setting for long-term textual viewing and editing.

One additional tactic that can be used to fight the effects of aging on vision is increasing the size and "trail" of the mouse and pointer icon. All operating systems allow for easily increasing the size of the mouse, and changing the color to make it more visible. Also, you can adjust the computer setting so that the mouse leaves a "trail" as it moves, making it easy to follow visually. On Windows systems one can set the mouse/pointer icon to flash when the Control Key is pressed, making it easy to find on a cluttered screen.

Often, aging computer users are faced with reading websites or documents designed by those who have little understanding of font legibility. Faced with a website in Comics Sans or one written in PLAYBILL, decreasing visual acuity can hinder understanding. Aging computer users need to be aware that many, if not most, applications allow substitution of a more legible font for one that is not readable. Research performed on message legibility has not come to a clear conclusion as to which factors make a font legible[16].

Our advice to the aging computer user is to find a set of fonts that allow for easy readability. Other features, such as font size and color, background, bold vs. non-bold, italicized fonts, etc., also affect readability. Different documents and types of computer use might require different fonts and color settings. Do not be afraid to experiment.

Physical Adaptations: Reduce Eyestrain

James Sheedy, director of optometric research at the Vision Performance Institute at Pacific University Oregon, put computer vision syndrome on the map two decades ago when he began to publish scores of studies on computers and vision[17]. It has been called a modern epidemic. Symptoms include eyestrain and fatigue, double or blurred vision, dry or irritated eyes, and aches in the head, neck and/or back (from improper head positioning). What distinguishes this from more generic eye complaints is that when the sufferer stops using a computer, the symptoms tend to disappear or greatly subside.

One reason that many users have "computer vision syndrome" is a simple one—many computer users either lack or have incorrect glasses for continuous, close-in computer viewing. A common mistake is to believe that bifocals will suffice. In fact, bifocals will often not only cause eyestrain, but due to the user constantly holding their head at a less-than-optimal angle, neck strain will also result[18]. A simple visit to an eye doctor can provide the computer user with glasses designed for computer use. It has been shown by an University of Alabama study that it is cost-effective for the employer to provide computer users with eye care and specific glasses to prevent eye fatigue, with a cost/benefit ratio of over 2:1[19].

Other possible adaptations to reduce eyestrain include[20]:

• Upgrade to glare-free lighting. Overhead fluorescent lights should be indirect, or have louvers to diminish the brightness of the light source. Avoid a high contrast between your computer screen and room lighting by lowering bright light sources and adding blinds to windows or adjusting the brightness of the screen. Task lighting can help illuminate text if necessary. The University of Alabama studies, above, have also shown that florescent lighting is far superior to incandescent bulbs.

• Place your monitor straight ahead, an arm's length away when you are sitting in front of it, where you can view the middle of the screen without tilting your head up or down. Position the monitor perpendicular to windows, and keep your screen clean to reduce blurred vision.

• Use corrective lenses that allow clear viewing of the screen. That might mean a special pair of glasses that you use just for the computer. (Bifocals and progressive lenses might cause you to tilt your head back to see, which can lead to poor neck posture.)

• Take regular breaks. Follow "the rule of 20s": Every 20 minutes, stand up, walk to a window if you have one, and look 20 feet away from your screen for at least 20 seconds. Note that such breaks can be productive: they are an ideal time to make phone calls, catch up on face-to-face meetings or review printed material.

• Blink often - it moistens the eyes. In one study, Sheedy[21] found that computer users' blink rate dropped 50 percent when they were staring at a monitor (from 15 per minute to seven and a half). This definitely contributes to dry eyes.

• Avoid squinting. This happens far more often than you may realize because you cannot see the screen clearly (or the screen is too bright) or because of glare. Another cause could be improper vision correction. All of these can lead to eyestrain.

• One way to reduce eyestrain is to occasionally "sooth your eyes." Rub your hands together briskly to create heat, then palm your eyes by placing the heel of your hands on your cheekbones and fingertips in your hairline. Without pressing on the eyeball, block out all light and allow the warmth to soothe the eyes. A good eye exercise — for everyone — is to imagine a large clock in front of you. Without moving your head or straining in any way, let your eyes trace a slow clockwise circle, then a counter-clockwise one. Close your eyes and rest them[22].

Physical Adaptations: Ergonomics and Physical Environment

Poor usage of the keyboard and mouse can lead to significant medical problems (e.g., carpal tunnel syndrome). Many computer users believe that switching to an ergonomic keyboard and an alternative pointing device such as a trackball or a trackpad will alleviate this problem. However, this is not universally the case[23]. In fact, many ergonomic keyboards simply change the musculoskeletal region exposed to risk, instead of eliminating hazardous postures. Regardless, it is generally accepted that an ergonomic keyboard minimizes the potential for carpal tunnel syndrome, even though there are not any universally accepted benefits. Proper posture and correct typing skills are most likely equally effective. It boils down to which type of keyboard enables the user to type faster and more accurately.

Alternative pointing devices likewise do not have clear advantages in terms of preventing strain or injury. Nevertheless, they have their place. Many computer users feel often that a trackball or trackpad is not as tiring as using a mouse, especially after a long period of use. However, there is a learning curve associated with these alternative pointing devices; do not expect computer users to become accustomed to them without a "break-in" period. Another common solution is to switch the hand that you use for the mouse. For example, it is not uncommon for some people to alternate from one hand to the other every month to alleviate any problems in that hand. Note, however, that there is a steep learning curve for a person who has used their mouse with their right hands for many years when they attempt to user the mouse with their left hand. Personal experience on the part of one of the authors (due to carpal tunnel syndrome) suggests that it takes several months for "wrong-handed mousing" to feel natural or be accurate. The author eventually learned to use one hand for the mouse, and another for the trackpad—and both now feel natural. It is possible that the brain is better able to adapt to separate hands for separate pointing devices, but a search of the literature has revealed no published evidence for this.

Along with optimum viewing distance for monitors, one also needs adequate space for keyboards and the mouse. The user should not be cramped in terms of elbow room or room to use the mouse. There should also be enough space for the user to use the keyboard correctly[24].

Although most people are well aware of the more common problems with keyboard ergonomics, there are many other areas to consider as well. For example, one should sit on an adjustable chair and raise or lower it until thighs are parallel to the ground. This helps alleviate potential knee problems with chairs that are too high or low. In addition, one of the easier solutions to neck pain, besides good posture, is to have the top of your monitors level or slightly lower than your eyes.

Last, as we age, our hearing tends to deteriorate, a condition aptly termed "age-related hearing loss[25]" occurs. The small, tiny speakers included in many laptops and desktop computers no longer generate the desired volume (or acoustic clarity) when such hearing loss occurs. Typically, a set of reasonably inexpensive speakers (approximately $25) is all that is necessary for the aging computer user to regain the ability to hear computer-generated audio clearly. If working conditions would make using speakers infeasible, a moderately inexpensive set of headphones ranging from in-the-ear headphones at $10 to over-the-ear higher-quality headphones at $40 will make listening to audio clearer, easier, and comfortable.

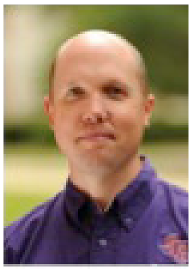
Summary

No matter how hard we try, developers grow older. There is no miracle fountain of youth that will stop you from aging. Fortunately, there is a silver lining to aging. Being older also means having more experiences and usually greater wisdom. Equipped with larger, crisper monitors and the greater experience and wisdom that you have can be used to make you even more valuable to your business as time goes on. The increased value of crystallized intelligence can easily offset the slight deterioration of fluid intelligence.

As a manager, do not expect your workforce to stay young. Increase the productivity of your existing workforce by adapting their environments to their needs. Do not expect the workers to adapt to physical changes—instead provide an environment that adapts to their individual needs. The important point is that computer technologies are tools to help your developers perform their job. Personalize your environment by adjusting settings, using different devices, and other adjustments to improve their performance, instead of letting the environment restrict performance.

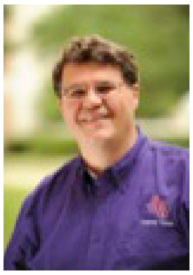
You should choose new technologies that will increase your efficiency (such as larger monitors). Finally, do not overlook the value of crystallized intelligence that will allow increased performance from computer users in spite of advancing age.

ABOUT THE AUTHORS



Robert Ball, Ph.D. is an assistant professor of computer science at Stephen F. Austin State University in Nacogdoches, Texas. Dr. Ball obtained his doctorate at Virginia Polytechnic Institute and State University. Previously, he worked at Pennzoil and NuSkin as a software developer. His current work has focused mainly on understanding how to increase older adult productivity with computers.

E-mail: ballrg@sfasu.edu
Phone: 936-468-2508
Department of Computer Science
Stephen F. Austin State University
Nacogdoches, TX 75962



David Cook is Associate Professor of Computer Science at Stephen F. Austin State University. He served 23 years in the Air Force, teaching computer science and software engineering at both the USAF Academy and AFIT. He also worked as a consultant to the STSC for 16 years. His fields of interest are software engineering, software quality, and verification and validation of large-scale modeling and simulations. His Ph.D. in computer science is from Texas A&M.

E-mail: cookda@sfasu.edu
Phone: 936-468-2508
Department of Computer Science
Stephen F. Austin State University
Nacogdoches, TX 75962

Professor Michael M Pickard is the current Chair of the Department of Computer Science at Stephen F. Austin State University. He holds a M.S. and a Ph.D. in computer science and a B.A. in mathematics from Mississippi State University. Before entering academia he had a 20 year career in the computer industry, including nearly eight years as a USAF officer. Software engineering is his principal area of interest.

E-mail: mpickard@sfasu.edu
Phone: 936-468-2508
Department of Computer Science
Stephen F. Austin State University
Nacogdoches, TX 75962

NOTES

1. There are various statistics on cosmetic surgery. The three primary sources that we used are from The American Society of Plastic Surgery, www.plasticsurgery.org, Plastic Surgery Resaerch.Info, <www.cosmeticplasticsurgerystatistics.com>, and general trends from The Guardian, <www.guardian.co.uk/news/datablog/2011/jul/22/plastic-surgery-medicine>.
2. P. Moore and C. Conn "Disguised: A True Story" (Word Books, 1985).
3. See for example, R. Zemke, C. Raines, and B. Flipczak. "Generations at Work: Managing the clash of Veterans, Boomers, Xers, and Nexters in Your Workplace," (AMACOM Books, 2000); L. Lancaster and D. Stillman, "When Generations Collide," (Harper Collins, 2002).
4. See for example, A. Guyton, and J. Hall, "Textbook of Medical Physiology. 10th ed.," (Saunders, 2000): 566-77; R. Watanabe, "The Ability of the Geriatric Population to Read Labels on Over-the-Counter Medication Containers," Journal of the American Optometric Association, Volume 65 (1994): 32-37.
5. R. Ball and J. Hourcade, "Rethinking Reading for Age from Paper and Computers," (International Journal of Human-Computer Interaction. Volume 27, issue 11, 2011), pp. 1066-1082.
6. R. Cattell "Intelligence: Its structure, growth, and action" (Elsevier, 1987).
7. R. Ball and J. Hourcade, "Rethinking Reading for Age from Paper and Computers," (International Journal of Human-Computer Interaction. Volume 27, issue 11, 2011), pp. 1066-1082.
8. R. Ball and C. North, "Realizing Embodied Interaction for Visual Analytics through Large Displays," (Computers & Graphics (C&G) Special Issue on Visual Analytics, Volume 31, issue 3, 2007): 380-400.
9. A. Sabri, R. Ball, S. Bhatia, A. Fabian and C. North, "High-Resolution Gamin Interfaces, Notifications and the User Experience," (Interacting with Computers Journal. Volume 19, issue 2, March 2007): 151-166.
10. A. Maslow "The Psychology of Science" (Gateway Editions, 1966): 5.
11. D. Burns "The Feeling Good Handbook" (Plume, 1999).
12. A study by (Klein, et al. 1992) D. Klein, et al.., "Vision, Aging and Driving: The Problems of Older Drivers," (Journal of Aging and Gerontology, Volume 47, Issue 1, 1992): 27 - 34.
13. For example, see R. Ball and C. North, "An Analysis of User Behavior on High-Resolution Tiled Displays." in Tenth IFIP International Conference on Human-Computer Interaction (INTERACT 2005), pp. 350-364; D. Tan, M. Czerwinski, and G. Robertson, "Large Displays Enhance Optical Flow Cues and Narrow the Gender Gap in 3D Virtual Navigation," Human Factors: The Journal of the Human Factors and Ergonomics Society (2006).
14. R. Ball and C. North, "An Analysis of User Behavior on High-Resolution Tiled Displays." in Tenth IFIP International Conference on Human-Computer Interaction (INTERACT 2005), pp. 350-364.
15. R. Ball, "Upgrading Human Performance, Not Computer Performance", Graziadio Business Report. Volume 13, issue 1 (January 2010).
16. Bix, L. (2002). The Elements of Text and Message Design and Their Impact on Message Legibility: A Literature Review. Journal of Design Communication, No. 4.
17. J. Sheedy. Focus on computer-generated eye problems. Occupational Health & Safety 64(6), 46-50, (1995)
18. E. Pascarelli M.D. and D. Quilter Repetitive Strain Injury: A Computer User's Guide. Wiley (1994)
19. K. Daum, et. al., Productivity associated with visual status of computer users. Optometry. 2004 Jan;75(1):33-47.
20. E. Pascarelli M.D. and D. Quilter (1994). There are many similar checklists available, but this checklist, from Quilter, can be found at <http://www.nextavenue.org/article/2012-01/your-computer-killing-your-eyes>
21. J. Sheedy, et. al., Blink rate decreases with eyelid squint. Optom Vis Sci. 2005 Oct;82(10):905-11.
22. There are multiple sources for advice on how to prevent eyestrain. The Occupational Safety and Health Administration has compiled one such list, available at <http://www.nextavenue.org/article/2012-01/see-light-prevent-eyestrain-while-computer>
23. M. Fagarasanu and K. Shrawan, "Carpal tunnel syndrome due to keyboarding and mouse tasks: a review," International Journal of Industrial Ergonomics (2003)
24. There are a number of resources available for the reader to check keyboard ergonomics. For example, <www.healthycomputing.com/office/setup/keyboard>
25. R. Patterson et al., "The deterioration of hearing with age: Frequency selectivity, the critical ration, the audiogram, and speech threshold," Journal of the Acoustical Society of America, Volume 72, Issue 6 (1982): 1788–1803.

Programming Will Never Be Obsolete

Andrew Mellinger, SEI

Abstract. We live in world that will always be full of problems. Changing conditions and advances in science and current solutions are constantly providing even more opportunities daily. While these areas may share similarities to previous problems, the essential fact that they have not been solved means that creativity is required to provide a new solution. It is this need for creativity that prohibits machine and algorithms from dealing with this issue and that we will need a programmer to translate these solutions into executable form.

Programming Is and Always Will Be Important

We have all heard the argument that programming will become obsolete. Notions like "it is a dead end career" or "salaries will drop" are constantly plaguing the viability of the field. A quick web or periodical search will return articles on the topic from at least as early as 1984, and there are new ones being posted every day. They range from scholarly articles such as, "Can fifth-generation software replace fallible programmers?" to modern blog posts that cut to the chase, "Is Programming Really as Future Proof a Profession as People Think?" [1] [2].

The issue is raised for a variety of reasons, some of which are honest and some are disingenuous. I prefer to focus on the genuine concerns of developers, technologists, and academics that the end of programming and their careers will be brought on by automating programming tasks or the end of a particular technology on which they depend. I will ignore disreputable claims that the problem can be solved by adopting a certain vendor's technologies or getting particular platform certifications.

Often, people will see a decline in a particular technology or method and will prophesize the fall of programming generally, rather than as it pertains to the specific technology. The need for programming may decline for programmers near the end of a specific technology's lifecycle, but the general technological challenges are moving targets, and therefore, we will always have new problems.

When discussing programming, some people are referring to the act of typing in the code, and some mean the entire software development lifecycle. This article includes all aspects of development and will use development (developer) and programming (programmer) synonymously. Programming, development, and engineering are highly related activities but focus on different dimensions of the overall software production process. The difference between these high level activities is the degree to which they directly interact with code. Programming is the activity that is closest to the code, while engineering is generally the farthest. Programming is where the developer picks and chooses from the available technologies, patterns, accumulated practices, techniques and their experience to "best" satisfy the complex interaction of requirements.

It is this fundamental interaction with the code that differentiates the actual act of programming from other activities. Programming should be not conflated with the physical act of typing, but equated with the "last mile" of actually coding, or expressing the intent in an executable language. Some would argue that this is simply a translation process, but for anyone who has worked on a project of substantial size, it is much more. In simple natural language translation the input and outputs of both are at the same semantic level. For example, if I am translating "My hovercraft is full of eels" from English to Swedish, I am trying to say the exact same thing in both languages. In programming there is a change of semantic level. For example, the requirement may be to "support undo" which implies a variety of user interface interaction points, interactive behaviors, and changes to storage semantics. One may argue that undo is a complicated concept and should not be handed to a "programmer" but in practice projects frequently hand problems of this complexity to a developer, or the person who is touching the code. Modern frameworks have a lot of infrastructure to support complex patterns like undo, but there are still a wide variety of decisions to be made by the developer with regards to the domain specifics.

Eras in Technology

Technologies rise and fall in popularity, and while they drive business growth they also require a tremendous amount of programming. New technologies arrive with a bang and drive the economy for some period of time through tooling, employment, and products. These periods, or "eras," vary in size, length, and overall impact. Eras overlap with those of other technologies such as different languages, software platforms, hardware architectures, peripherals, and development methods that draw an incredible amount of innovation. Consider the iPhone, which was introduced in 2007 and opened up new economic and technological markets. At that time there was a huge demand for Objective-C/Cocoa programmers and people who understood the special nuances of mobile device interaction and their interfaces. The iPhone impact had a ripple effect through the tech industry and ushered in Android technology, which introduced an increased demand for Android/Java programming. Then the tablet arrived and created a tablet/phone hybrid tsunami.

During each technological era we see cutting edge technologies move from the inventors and innovators to early adopters and eventually adoption by the masses. Most successful eras possess similar qualities such as a wealth of new ideas, financial investment, fierce competition, and general uncertainty. How does a developer live through this cycle? We are bombarded by a wealth of new technologies touted by vendors, researchers and volunteer communities. Which do we choose to learn? What do we follow? It is impossible to review, much less understand every language, framework, tool or platform that arrives and we need to choose some that keep us fresh and might help our current job. At some point in each technological era, the cutting edge becomes not so sharp and the leaders are identified. This is the time period that makes it easier to choose which technologies you should learn and adopt. Eventually, the era progresses to the point where the technology area enters the mainstream. This is when we typically see the publication of books on the subject, and finally the emergence of the "standard" technologies, protocols, or methods.

Over time these technologies become commonplace when point-and-click tools, or off the shelf packages that are suitable for a vast majority of the instances. As is a programmer's nature, when we see something that is "routine" we write a script or app or framework to do it faster, cheaper, and better. This is when you will see a decrease in the need for the specialized skills and training of a programmer. However, this will also usher in its own set of doomsayers and charlatans. What is becoming "obsolete" or in less demand is the need for a particular set of skills, not for technology problem solvers.

The Programmer's Role

When I interview people for programming positions, I divide them into two categories: programmers who focus on a particular technology and programmers who focus on the underlying principles of technology. A programmer that advertises themselves as an "insert-favorite-technology-here developer" instead of as a "software developer" is more likely to learn one or two skills the market needs and work exclusively within those roles. I refer to this type of programmer as a "technician" as opposed to a "general purpose developer." The technicians are often the people who argue that their favorite technology is the solution to all of your problems. While they may be masters at that technology (or a handful of them), their fate is inevitably tied to it. Do not get me wrong, these can be tremendously creative, talented, and smart people, but they have a very limited focus. When that technology declines they will find themselves having difficulty finding work and will blame it on the fact that "programming is dying" when in reality they have not stayed relevant.

General purpose developers are not tied to a technology, they have tied to technology. They get bored working with just one technology, which is good. This drives them to attempt to automate things and make technology cheaper, faster, and better. These developers are ready to move to new languages or platforms as they become available because they are not focused on one technology. Development requires decision making and creativity, which are two things we cannot automate. Granted, general developers may become focused (sometimes obsessively so) on a technology for a while, but eventually find the need to tie their work to a general computing and technology problem. The ability for programming generalists to be creative and apply fundamental programming principles to build new technologies is the cornerstone that continues to make them cutting-edge and essential to business growth.

Fundamentally, computers are good at doing what we tell them to do. This means that someone must understand what we want them to do in the first place. A software developer's fundamental job is to take knowledge and make it "executable" or "actionable." The job also requires discovery of this knowledge through requirements definition, usability studies, domain analysis and prototyping. Software architecture, design, and coding all require a significant amount of analysis, reasoning, and decision making. Consider that so many companies want their developers to provide "revolutionary" products, and we can see that creativity will be a requirement for years to come.

Essentials of Programming

We will not run out of problems to solve. Whether they are core research problems or applying some set of solutions to a particular job, we need to look at what the essential qualities of programming are and why they will persevere. Even if we create a solution to a problem, the solution itself is likely to create new problems.

In "No Silver Bullet—Essence and Accident," Fred Brooks argues that software development is so challenging that it will require human intellect for a long time due to four fundamental qualities: complexity, conformity, changeability and invisibility [3]. These qualities have not changed since he wrote the article over 25 year ago, and do not seem likely to change. It is these same qualities that we are trying to use technology to solve, but it is technology that keeps moving the problem ahead of our solutions.

On the implementation side alone, as we continue to discover and learn more, we will always need someone to translate that knowledge from the domain into something executable. We will always need someone to fill that gap as there will always been that point where a person can make an executable representation but where it is not routine enough to automate. We will always be encountering new problems and the sheer nature that they are new problems means they have not been solved. Certainly, many problems in that class may have been solved by many long nights by developers, but not the general problem itself. Even when reusable patterns exist such as a framework, technicians will be required to encode a specific instance such as a particular website or cloud instance for that problem.

When we take all of this into consideration, programming as a creative work will cease to be needed when we have automated all other creative knowledge work. We are more likely to make lawyers, insurance salesman, or politicians obsolete before programmers. One could argue that in the very far future once we have discovered everything and can finally automate the very last thing, that last job will be for a programmer.

Being a Developer in the Future

So what will programming be like in the future? At its core, it will be like it is now. Developers will work to understand the domain, do general problem solving and knowledge creation and then instruct machines on how to execute these solutions. They will need all the skills of the general developer and some understanding of their domain. And they will need to be able to learn and adapt. Marc Andreessen argued in "Why Software Is Eating the World" that as more and more things include a software component, general software developers will always have new problems to tackle [4].

What are the next possible technology areas? A quick glance at the Gartner Hype Cycle can help us prepare [5]. Mobile and cloud technologies are well underway, but that space is very broad and deep with tremendous needs of usability, security, and big data. We have barely scratched the surface with autonomy, ubiquitous computing and the broad application of 3D printing; much less the ones further out such as nanotechnology or biotechnology. Some of these are not computing problems, at least how we know it now, but will certainly require "programming" of some sort. One can peruse modern science fiction to see how a programmer's world might be different in the years to come.

We live in a world that will always be full of problems. Changing conditions and advances in science and current solutions are constantly providing even more opportunities daily. While these areas may share similarities to previous problems, the essential fact that they have not been solved means that creativity is required to provide a new solution. It is this need for creativity that prohibits machine and algorithms from dealing with this issue and that we will need a programmer to translate these solutions into executable form. On the other hand, the specific technologies will change as we routinize these tasks and climb the abstraction ladder. Because of this, specific programming and programmers may become obsolete, but new problems will always require new solutions and general programmers to implement them.

Disclaimer:

ABOUT THE AUTHOR

Andrew Mellinger is a member of the technical staff at the SEI. His passion for computing started at age 12 when he wrote his first commercial piece of software for the company where his father worked. He currently focuses on data intensive scalable computing, security informatics, cloud computing, and adaptive and heterogeneous architectures.

Software Engineering Institute
Carnegie Mellon University
4500 Fifth Avenue
Pittsburgh, PA 15213-2612
Phone: 412-268-5800
Toll-free: 1-888-201-4479
<www.sei.cmu.edu>

REFERENCES

1. Philips, R. Can fifth-generation software replace fallible programmers? Computerworld, v 18, n 29, 1D/27-30, 16 July 1984
2. Perry, Jon; Kupper, Ted Is Programming Really as Future Proof a Profession as People Think? Accessed November 2013 http://declineofscarcity.com/?p=2557
3. Brooks, Frederick P. No Silver Bullet: Essence and Accidents of Software Engineering. IEEE Computer, vol. 20, pp 10-19, 1987 <http://www.gartner.com/technology/research/methodologies/hype-cycle.jsp>
4. Andreessen, Marc. Why Software Is Eating The World August 2011. <http://online.wsj.com/news/articles/SB10001424053111903480904576512250915629460>
5. Gartner, Inc. Last accessed November 2013

Identifying Good Independent Variables for Program Control

Bob McCann, Lockheed Martin Aeronautics

Abstract. Many of the behaviors and adverse outcomes that we see in software-intensive programs are the result of "misaligned incentives" between the goals of the individuals involved and those of the larger organization. These interact and play out in recurring dynamics that are familiar to both software developers and managers, but which are still poorly understood. By characterizing the forces within these dynamics explicitly in the form of the "acquisition archetypes" described in this paper we can come to understand the underlying mechanisms that cause these problems, and identify mitigations to help mitigate and prevent them.

Abstract

There is an important distinction between program control and program tracking. Control is predictive and proactive using a causal production model that clearly identifies input variables. Tracking is outcome based and decisions based on tracking are reactive and uses output measures. This article analyzes one such causal model and uses it to identify the limits of control (what you can and cannot accomplish with the identified control). The purpose here is to demonstrate how to pick a good control variable from the set of variables derivable from the model.

Introduction

To set expectations, the purpose of this article is to design a better brick not to build a subdivision. In what follows, it is important to distinguish control variables from tracking and oversight metrics. Tracking and oversight metrics are always after the fact, lagging indicators. Control variables are always predictive variables—specifically ones that are open to adjustment. It is also important to distinguish causal models from empirical ones. Causal models make a clear quantitative link between causes and effects while empirical models show general trends without regard to causality and detailed understanding of production processes. Causal models are useful for controlling individual process flows. Empirical models are generally useful for bounding the cost and schedule estimates for proposals before the detailed production processes have been chosen or designed.

Given what we know about delivering products and services using defined, repeatable processes[1], how do we pick the best controls for the project? Surprisingly, the answer comes from basic mathematical concepts. Answer: Pick the variables with the best condition numbers, but what does that mean? The idea is that the process output should be single valued and behave smoothly with respect to small changes in the control variables; in short the process output measure should be a smooth mathematical function of the inputs[2]. This article will demonstrate the application of that principle to a simple causal cost-benefit model.

For instance, if there is a relationship between quality, cost and schedule, then which one is the best independent variable to use in managing the contract? There are three choices:

- **cost as an independent variable**
- **schedule as an independent variable**
- **quality as an independent variable**

Which is the best to use? How do we answer that question?

In what follows a specific simple model is explored to demonstrate how to answer this question. The concepts generalize for more complex models. The chosen model is causal rather than empirical. That is an important distinction because causal models more easily identify how specific actions drive results in the overall project performance. That makes clear that program decisions are controlling the program by directly addressing causes to produce predictable results.

Mathematical Foundations

For linear functions, the choice of which variable to use is a matter of convenience. Not so for curves and more complex functions. It is important that the dependent variable be well behaved when represented as a function of the independent variables. First of all, one should pick an independent variable for which the dependent function is single valued, and secondly the dependent variable should be well conditioned with respect to the independent variable—it should change smoothly and proportionally to small changes of the independent variable. For one-dimensional functions of a single variable, the answer is described by the slope of the function; the function should not have spikes and other singularities near the point of optimum performance.

The example model analyzed here is a simple model for a single sequential process flow with a series of developmental tasks that affect the quality and cost of the end product in a traceable way. For more complex situations, it is necessary to consider the condition number.[3] The condition number is the ratio of the maximum to minimum eigenvalues of the matrix of the first derivatives of the dependent functions with respect to the independent variables—the Jacobian Matrix. (This is a direct application of the inverse function theorem and the definition of condition number for linear systems.)

Schedule is too complex to discuss at the necessary level of detail; it would easily take a whole book, so it will not be treated directly here. Heuristically, for small enough changes in the independent variables, schedule can behave simply. To see this, consider a staffing curve consisting of intervals with constant staffing. Fixed costs will be irrelevant to the marginal analysis that follows. Since marginal cost for applied labor is roughly schedule times labor rate, we can conclude that Cost and Schedule have a more or less piecewise linear relationship and are thus more or less interchangeable in a sufficiently small neighborhood of the operating point. Of course that does not mean that schedule is easy to manage over the life of the project because that simplification does not help when changes become large or discontinuous such as when the workflow on a PERT network shifts from one critical path to another. In general, it takes sophisticated analysis using a robust professional quality-scheduling tool such as Primavera or Artemis to manage the critical path and the most likely near-critical paths of the project's PERT Network. That complexity often obscures the simple relationship that follows from looking at the relationship between cost and quality for individual process flows.

How Does Quality Qualify?

What about quality? With respect to quality, in the example model, marginal cost can be expressed as a linear function of defect injection rate[4] and a non-linear function of document review preparation rate.[5] When plotted as a function of preparation rate, it is clear that the function has a single minimum[6], see Figure 1. This choice meets the criteria identified above.

However, expressing quality as a function of cost is generally double valued and has a point of infinite slope at the cost minimum (the Functional inverse near points with a vertical slope is very poorly conditioned). Clearly, per Figure 2 and starting from the right, there is a region with two possible operating points—one with high quality and one (more likely) with lower quality. Then there is a point with a single optimal operating point and vertical slope, and last there is a region with no solutions. Clearly using cost as the independent variable creates a situation that is poor from the program control perspective—a multivalued relationship on the high cost side of the optimum cost point, a point of infinite condition number at the point of optimum performance, and a region with no stable solution on the lower side of the cost optimum. Programs managed by cost control while under cost pressure have a clear risk of driving off the cliff into chaos.

Please remember that program control is not the same as progress tracking. For example using Earned Value to track progress against budget and Earned Schedule to track progress against schedule both clearly have value but cannot be effective program controls at the task level on individual process flows because Earned Value is a non-causal tracking model. Managing the quality control variable (document review preparation rate) is a much more effective approach to program control for individual process flows at the task level.

Conclusion

Thus, for the example cost-benefit model for a single process flow, we can conclude that of the three discussed, cost, schedule and quality, the best independent variable is quality as represented by the document review preparation rate. This is an example of what kind of analysis supports the decision of what control parameter to use proactively (leading indicator) in program control. The basic requirement is a quantitative, causal model with measurable, adjustable parameters for the known causes of variation rather than an empirical descriptive scaling model. The ones with the best functional behavior are the ones to use. Heuristically in the following figures, the right answer smiles and the other turns everything on its side and sticks its nose into chaos where it doesn't belong.

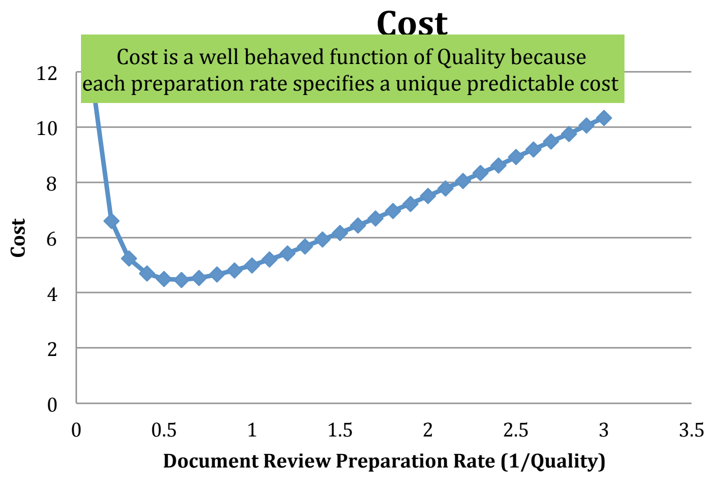


Figure 1 Cost as a Function of Document Review Preparation Rate (Dimensionless Scales)

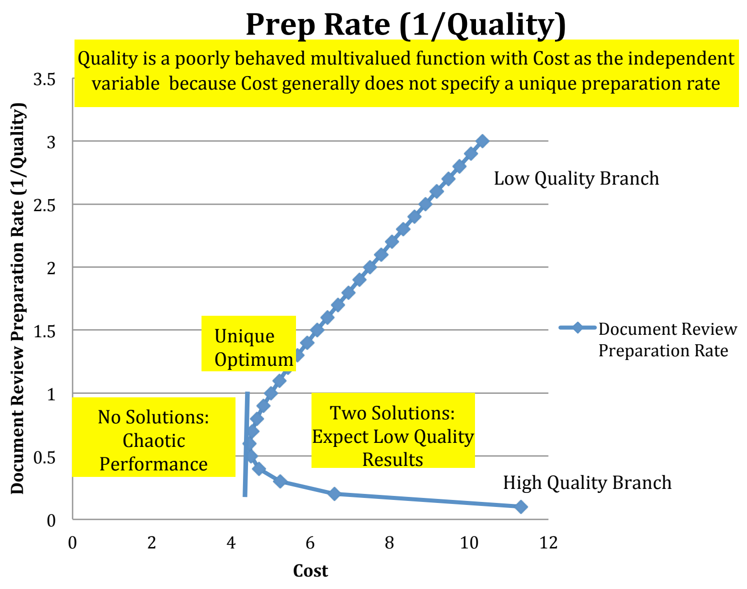


Figure 2 Document Review Preparation Rate as Related to Cost (Dimensionless Scales)

Disclaimers:

Copyright 2013 Lockheed Martin. Non-Export Controlled. Releasable to Foreign Persons. Non-Proprietary Information.

ABOUT THE AUTHORS

Bob McCann is a staff systems engineer at Lockheed Martin Aeronautics in Fort Worth, Texas. He is currently an Institute of Electrical and Electronics Engineers Certified Software Development Professional and has nearly 20 years of experience in computational physics and high performance computing including nine years at Princeton Plasma Physics Laboratory working in the U.S. Department of Energy-controlled fusion program, as well as about 10 years experience in design and development of relational databases of various kinds. Mr. McCann has served as a member of the Lockheed Martin IS&S Metrics Process Steering Committee and currently works on improving systems engineering processes, methods, and metrics. He has also studied Aikido since 1982 and has taught Aikido for 20 years.

He has a Bachelor of Arts in physics with a concentration in mathematics from Shippensburg University, a Master of Science in physics from University of Maryland, a Master of Science in computer science from Southwest Texas State University, and a Master of Science in computer systems management/software development management at the University of Maryland University College.

Lockheed Martin Aeronautics
P.O. Box 748, Mail Zone 2893
Fort Worth, Texas 76101
Phone: 817-935-4037
Fax: 817-935-5272
E-mail: bob.mccann@lmco.com

NOTES

1. Process based organizations can be very predictable and deliver high quality products and services at low cost. Such organizations can use an assessment tool such as the CMMI Institute's Capability Maturity Model Integrated to understand how well they are organized and how effective that structure can be (can do everything it needs to do). However, measuring performance is the organization's responsibility and is not an explicit part of the assessment model. <http://cmmiinstitute.com>
2. See <http://en.wikipedia.org/wiki/Function_(mathematics)>: In mathematics <http://en.wikipedia.org/wiki/Mathematics>, a function <http://en.wikipedia.org/wiki/Function_(mathematics)#cite_note-1> is a relation <http://en.wikipedia.org/wiki/Binary_relation> between a set <http://en.wikipedia.org/wiki/Set_(mathematics)> of inputs and a set of permissible outputs with the property that each input is related to exactly one output.
3. Please follow the hyperlinks to see a full development of the mathematical concepts where that understanding is a bit rusty.
4. Ron Radice, "High Quality, Low Cost Software Inspections," Paradoxicon Publishing, 2001. This 14 chapter, 478 page book provides a firm foundation for using semi-formal review of intellectual work products, documents in general and code in particular, as an effective program control.
5. See p. 32 for a derivation of the cost function: Robert T. McCann, Cost-Benefit Analysis of Quality Practices <http://www.amazon.com/Cost-Benefit-Analysis-Quality-Practices-Robert/dp/0769546595/ref=sr_1_2?s=books&ie=UTF8&qid=1332876341&sr=1-2> Note that this IEEE Ready Note is a compilation and extension of three Crosstalk articles:

- Robert McCann, "How Much Code Inspection is Enough?" CrossTalk, July 2001
- Robert McCann, "When is it Cost Effective to use Formal Software Inspections?" CrossTalk, March 2004
- Robert McCann, "The Relative Cost of Interchanging, Adding, or Dropping Quality Practices," CrossTalk, June 2010

6. Ibid, Figure 3-2, p. 34

"If it passes test, it must be OK"

Common Misconceptions and The Immutable Laws of Software

Girish Seshagiri, Advanced Information Services Inc.

Abstract. As the saying goes, "If it passes test, it must be OK." Common misconceptions about managing software inhibit changes to the way software projects are planned, audited and assured for cost, schedule, and quality performance. This article describes the immutable laws of software development as articulated by SEI Fellow Watts Humphrey and based on the author's considerable professional experience in managing software technical teams. The author describes the impact of each of the immutable laws on an organization's ability to deliver very high quality software solutions on a predictable cost and schedule. The author provides data from his company's projects to illustrate many of the laws.

Introduction

After retiring from IBM, Watts Humphrey made an "outrageous commitment" to change the way software applications development services are acquired, sold and delivered. In addition to the CMM®, Watts was the principal architect of the Team Software Process (TSP) and the Personal Software Process (PSP) [1, 2, 3]. My company AIS was one of the early adopters of TSP and PSP. I was fortunate to work closely with Watts and built AIS's software development business making quality the number one goal.

After listening to many of Watts's presentations, and augmented by my personal experience in managing more than 200 software technical teams, I compiled a list of the immutable laws of software development. In this article, I discuss the implications of the laws and what acquirers, development management, and software teams need to be aware of to ensure consistent delivery of very high quality software systems and services on a predictable cost and schedule. I illustrate many of the laws with examples from AIS projects.

Software Engineering's Persistent Problems

Software Engineering like other engineering professions has had a beneficial impact on society. Arguably, the high standard of living in today's interconnected world is not possible without advances in software and software engineering. And yet there is ample evidence to suggest that software engineering as a profession has not been able to solve major persistent problems:

1. Exponential rise in cybersecurity vulnerabilities due to defective software
2. Unacceptable cost, schedule, and quality performance of Enterprise Resource Planning (ERP) and legacy systems modernization projects
3. Cost of finding and fixing software bugs (i.e. scrap and rework) as the number one cost driver in software projects
4. Arbitrary and unrealistic schedules leading to a culture of "deliver now, fix later"
5. Inability to scale software engineering methods even for medium size systems
6. Lack of understanding of the impact of variation in individual productivity
7. Absence of work place democracy and joy in work

Unless the software engineering professional community begins to systematically address these persistent problems, costs and risks to society will continue to increase [4].

The Immutable Laws of Software Development

Part of the reason for the persistent problems can be attributed to common misconceptions about managing the software work [5]. Organizations are either unaware of or are not willing to change practices to deal with the immutable laws.

I have listed some of the immutable laws and described the impact of each of the immutable laws on an organization's ability to deliver very high quality software solutions on predictable cost and schedule. Where applicable, I have provided data from our company's projects to illustrate many of the laws.

- **The number of development hours will be directly proportional to the size of the software product.**

While this is obvious, many projects do not estimate the size of the product before making a commitment for cost, and schedule. The implication of this law is that if an organization does not maintain a history of previous projects including the size of the product delivered and the effort in staff hours, the organization will make cost and schedule commitments with no relationship to the organization's historic capability. The cost and schedule commitment will be a guess based on the organization's desire to capture the business and not on what the organization can actually deliver. Which leads to the next law.

- **When acquirers and vendors both guess as to how long a project should take, the acquirers' guess will always win.**

In the beginning, neither the customer nor the developer knows how big the project is or how long it should take and at what cost. As Watts used to point out tongue in cheek, customers want their product now at zero cost. Customers usually have to deal with time-to-market pressures and they require the product in time frames that are arbitrary and unrealistic for the software team to produce a product that works. The developers now have a choice to make. They can try to guess what it would take to win the business. Or as rational management would require, elicit enough of the project requirement to be able to make a conceptual design, estimate the size, and use organization historic data to predict development time and cost. The TSP institutionalizes this behavior in the TSP team launch process in which all the developers participate in estimating and planning the project. The result is that teams make realistic and aggressive commitment that the team can meet. The implication of this law is that when faced with arbitrary and unrealistic schedule pressures, developers should have the skills to make a plan before making the commitment and the conviction to defend it. Otherwise, the customers' arbitrary and unrealistic schedule demand will become the team's commitment. Management should trust the team to develop an aggressive and realistic schedule, and not commit teams to a date that the team cannot meet. This leads to the next law.

• **When management compresses schedule arbitrarily, the project will end up taking longer.**

It is unfortunate that otherwise rational managers do not realize that the defect potential in a project increases disproportionately to schedule compression as many studies have shown. In one study of data from a large number of projects, a 20% schedule compression had the effect of increasing defects during development by 66%. [6]. The logical reason is that when teams do not have the time to do the job right, they end up skipping the quality steps and try to meet an impossible schedule in a code and test mode which ends up taking longer. This leads to the next law.

• **When poor quality impacts schedule, schedule problems will end up as quality disasters.**

This is a classic pattern in major software project failures. For instance, in the case of Healthcare.gov, one can speculate that the contractor teams were working to meet a deadline they knew was impossible to meet. The teams probably did not employ the quality practices they knew they should use. (In fact, some of the contractors were appraised at high CMMI® Maturity levels.) Instead they probably went through increasingly long cycles of code, test, and rework. Because the amount of rework due to poor quality is unpredictable, the schedule problem gets progressively worse. The team was forced to deliver poor quality product on the committed date, thus turning the schedule problem into a world famous quality disaster. Healthcare.gov is not the first such spectacular software project failure, nor will it be the last, as seen in the next law.

This is also borne out by AIS's early history from 1988 – 1992. The company was not profitable because our projects were not predictable. The projects always seemed to be on schedule through code complete and before the start of integration, system, and acceptance tests. Due to the poor quality, teams spent significant amounts of time in test and rework. People worked long hours, and heroic efforts were needed to deliver on the committed date. The customer acceptance test phase was not a

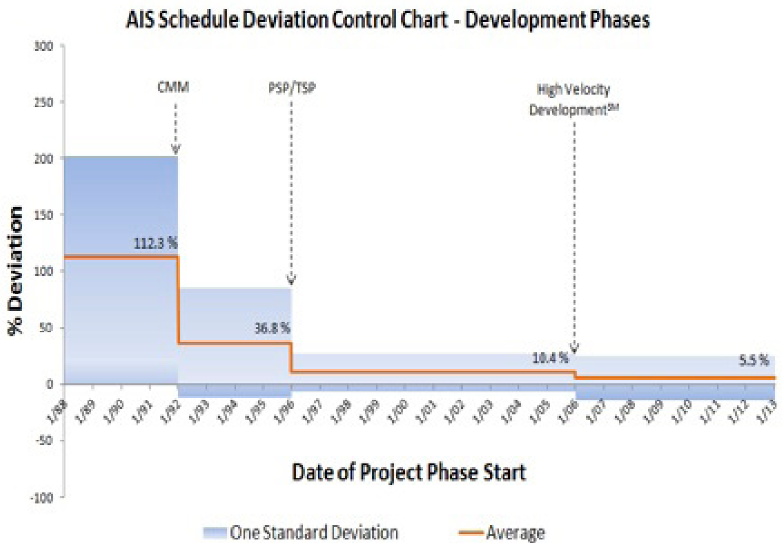


Figure 1: AID Schedule Deviation Control Chart – Development Phases

positive experience for either the customer or the team.

I realized that we had to change the way we managed the software work. What we needed was constancy of purpose with quality as the number one goal. Shown below is the schedule performance of AIS teams due to the improvement initiative I sponsored in 1992 based on the Capability Maturity Model (CMM) and later the TSP/PSP [7].

• **Those that do not learn from poor quality's adverse impact on schedule, are doomed to repeat it.**

The state of software practice will be much better for cost, schedule, and quality performance if only the c-level executives realize that poor quality performance is the root cause of most software cost and schedule problems. Remember SAM.gov, USAjobs.gov, and (ThriftSavingsPlan) TSP.Gov? These were noteworthy for cost and schedule overruns, the defects encountered in production and the long time it took to fix them, greatly inconveniencing the users of these applications. The government was doomed to repeat the experience in Healthcare.gov. This is not to single out government IT projects. Just that government projects get adverse publicity when they fail. It is probably not unreasonable to speculate that the commercial world is not immune to such quality disasters as documented in reports such as the Chaos report [8].

• **The less you know about a project during development, the more you will be forced to know later.**

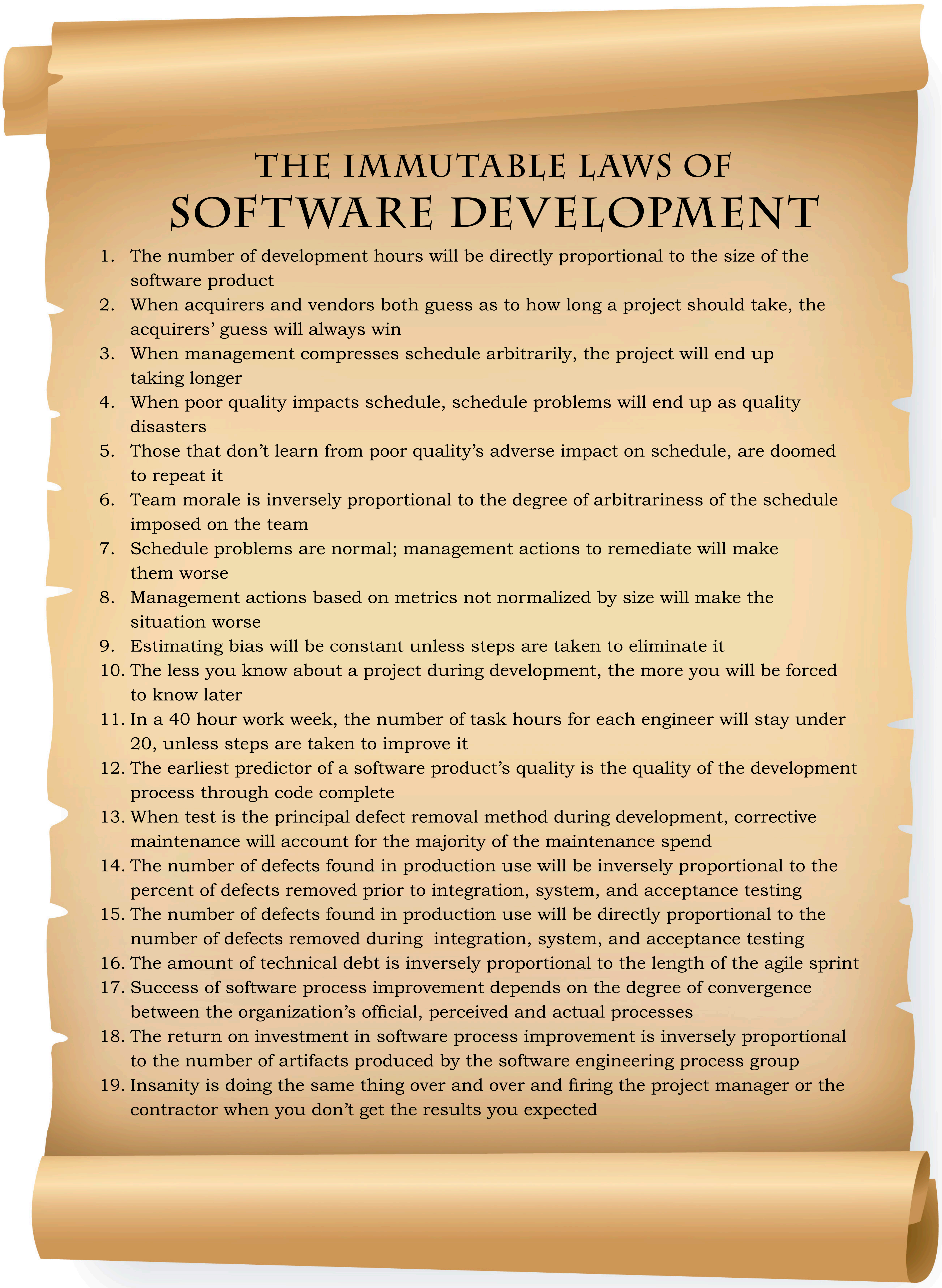
The implication of this law is that project teams need precise, accurate and timely information throughout development, to consistently deliver very high quality products on predictable cost and schedule. As Fred Brooks pointed out "Projects get to be one year late, one day at a time." When projects rely on the monthly status report as the only means of communicating what is happening in the project, they do not know enough to take timely corrective actions. When those projects fail, management relies on postmortems and audits to find out what went wrong.

In modernizing one of the largest databases in government, an AIS team collected and reported precise and accurate data in the weekly team status meeting. The team reviewed the project's documented goals weekly to make sure the team is on track to meet them. The team also reviewed the status of risk mitigation actions on the top 5 or 7 risks. The team made decisions weekly based on performance metrics that matter, including but not limited to plan vs. actual data on staff hours, earned value, defects injected, defects removed, and efficiency of early defect removal through personal reviews and inspections.

In many projects, one of the major causes for schedule slippage is because team members' actual hours on task are less than planned hours, which leads to the next law.

• **In a 40 hour work week, the number of task hours for each engineer will stay under 20, unless steps are taken to improve it.**

In estimating project schedules, teams typically do not consider the hours spent by team members on non-project tasks. In many organizations, the actual number of hours devoted to project tasks is on average less than 20 hours in a 40 hour work week. The implication of this law is that only management can take actions to improve the number of weekly task hours by providing improved office layout, minimizing



THE IMMUTABLE LAWS OF SOFTWARE DEVELOPMENT

1. The number of development hours will be directly proportional to the size of the software product
2. When acquirers and vendors both guess as to how long a project should take, the acquirers' guess will always win
3. When management compresses schedule arbitrarily, the project will end up taking longer
4. When poor quality impacts schedule, schedule problems will end up as quality disasters
5. Those that don't learn from poor quality's adverse impact on schedule, are doomed to repeat it
6. Team morale is inversely proportional to the degree of arbitrariness of the schedule imposed on the team
7. Schedule problems are normal; management actions to remediate will make them worse
8. Management actions based on metrics not normalized by size will make the situation worse
9. Estimating bias will be constant unless steps are taken to eliminate it
10. The less you know about a project during development, the more you will be forced to know later
11. In a 40 hour work week, the number of task hours for each engineer will stay under 20, unless steps are taken to improve it
12. The earliest predictor of a software product's quality is the quality of the development process through code complete
13. When test is the principal defect removal method during development, corrective maintenance will account for the majority of the maintenance spend
14. The number of defects found in production use will be inversely proportional to the percent of defects removed prior to integration, system, and acceptance testing
15. The number of defects found in production use will be directly proportional to the number of defects removed during integration, system, and acceptance testing
16. The amount of technical debt is inversely proportional to the length of the agile sprint
17. Success of software process improvement depends on the degree of convergence between the organization's official, perceived and actual processes
18. The return on investment in software process improvement is inversely proportional to the number of artifacts produced by the software engineering process group
19. Insanity is doing the same thing over and over and firing the project manager or the contractor when you don't get the results you expected

Figure 2: The Immutable Laws of Software Development

number of meetings etc. But the engineers have to record their time accurately including interruptions, to make management aware of low task hour utilization and the causes. In AIS projects, PSP trained engineers record time precisely and accurately and report task completions and earned value weekly.

- **The earliest predictor of a software product's quality is the quality of the development process through code complete.**

Software products are usually built from a large number of small components that are individually designed, coded, and tested. The PSP enables the engineers to build very high quality components through personal reviews and team inspections of the component's design and code artifacts. PSP trained engineers compile data on their personal process by recording size, time, and defect data on the components they build. By analyzing the component development process data, teams can determine the likelihood of the component having defects in downstream integration, system, and acceptance testing. The adverse impact on project schedule due to test and rework cycles in integration, system, and acceptance testing can be estimated before integration testing begins. AIS teams have a goal of more than 90% of the components to be error-free in integration, system, and acceptance testing. The impact of this law is that putting poor quality products into test will have adverse impact on the project's schedule and cost.

- **When test is the principal defect removal method during development, corrective maintenance will account for the majority of the maintenance spend.**

The implication of this and the following two laws is that putting poor quality product into test and relying solely on test for defect removal, has adverse cost implications beyond development. The biggest consequence is that as more defects are found in production use, organizations spend a very high percentage of the maintenance dollars in fixing bugs (i.e. corrective maintenance) instead of spending for the more beneficial enhancements and new features (i.e. perfective and adaptive maintenance). According to Watts, one of the software misconceptions is "if it passes test, it must be OK" [5].

- **The number of defects found in production use will be inversely proportional to the percent of defects removed prior to integration, system, and acceptance testing .**
- **The number of defects found in production use will be directly proportional to the number of defects removed during integration, system, and acceptance testing.**

The impact of these two laws is that early defect removal through personal reviews and team inspections, will result in high quality product (smaller percentage of defects remaining in the product) going into integration, system, and acceptance test which in turn will result in even higher quality product going into production. Conversely, putting a poor quality product (majority of defects remaining in the product) into integration, system, and acceptance test will result in excessive unplanned rework. What comes out of test will be a patched up product which in production use will uncover more defects to fix, thus consuming most of the maintenance dollars for fixing and keeping it running.

- **Success of software process improvement depends on the degree of convergence between the organization's official, perceived and actual processes.**

In every organization, there are usually three processes:

1. The official process, usually designed by the organization's software process engineering group, which describes the process the project teams should follow in their software projects.
2. The perceived process, which is what the software teams think how they do software work.
3. The actual process, which is how the teams actually work.

The implication of the law is that if the organization standard process is very different from the way the projects actually work, improving the standard process will be of little value. Project teams will continue to work the way they have in the past. In AIS, when we launched the continuous process improvement initiative, we first documented how the software teams were actually doing the software work. We used Watts Humphrey's Managing the Software Process book to establish a common vocabulary of process and process improvement. We empowered the engineers to make lots of small changes to the process by submitting simple but effective Process Improvement Proposals (PIPs). To-date AIS engineers have submitted more than 1400 PIPS of which more than 900 have been implemented. External SEI-authorized lead appraisers have appraised AIS's process maturity capability at CMMI Maturity Level 5 in 2007 and again in 2010 [7].

- **The return on investment in software process improvement is inversely proportional to the number of artifacts produced by the software engineering process group.**

The implication of this law is that if the process artifacts are produced by the software engineering process group and not the development teams, the artifacts may have little or no relationship to the actual work being done. The organization may pass maturity level appraisals without ever changing engineering behavior. Such organizations seldom produce very high quality products on predictable cost and schedule.

- **Insanity is doing the same thing over and over and firing the project manager or the contractor when you don't get the results you expected.**

This is a variation on the oft-used definition of insanity. The implication is that while people are extremely important, changing the people without changing the way the software work is managed is not likely to produce the expected results.

Conclusion

The relentless pressure to achieve a first-to-market advantage, has had the unfortunate side effect of developers more focused on meeting unrealistic schedule commitments than producing high quality software. We now have "deliver now, fix later" software development culture [9].

If the senior executives of software organizations understand the immutable laws and their impact, they will initiate the changes that are needed to consistently produce very high quality software on a predictable cost and schedule.

Disclaimer:

CMMI® and CMM® are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University

ABOUT THE AUTHOR

Girish Seshagiri is a globally recognized subject matter expert and thought leader in software assurance, software quality management, software process improvement, and modern methods of managing knowledge work. He is a reputed conference speaker, coach, and instructor. He is the executive sponsor of AIS's continuous process improvement resulting in the company's receiving IEEE Computer Society Software Process Achievement Award and Capability Maturity Model Integration (CMMI) Maturity Level 5 certification. He is the author of the white paper "Emerging Cyber Threats Call for a Change in the 'Deliver Now, Fix Later' Culture of Software Development."

Girish has an MBA (Marketing), from Michigan State University.

E-mail: girish.seshagiri@advinfo.net
Phone: 703-426-2790

REFERENCES

1. Humphrey, Watts S. PSP: A Self-Improvement Process for Software Engineers. Addison-Wesley Pearson Education, 2005.
2. Humphrey, Watts S. TSP: Leading a Development Team. Addison-Wesley Pearson Education, 2006.
3. Humphrey, Watts S. TSP: Coaching a Development Team. Addison-Wesley Pearson Education, 2006.
4. Seshagiri, Girish "Is the Two-Week Agile Sprint, the Worst Software Idea Ever? - Management Issues in Software Assurance and Information Security." CSIAC Webinar, October 30, 2013.
5. Humphrey, Watts S. Managing the Software Process. Addison-Wesley, 1989.
6. Donald M. Beckett and Douglas T. Putnam. "Software Quality, Reliability, and Error Prediction." STN 13-1 (April 2010)
7. Seshagiri, Girish. "High Maturity Pays Off. It is hard to believe, unless you do it." CrossTalk (January/February 2012)
8. CHAOS Manifesto 2013: Think Big, Act Small. The Standish Group International Inc.
9. Seshagiri, Girish. "Emerging Cyber Threats Call for a Change in the 'Deliver Now, Fix Later' Culture of Software Development." White Paper (September 2013)

Upcoming Events

Visit <http://www.crosstalkonline.org/events> for an up-to-date list of events.

The Blockbuster Conference on Software Testing and Analysis & Review
May 4-9, 2014
Orlando, Fl
http://stareast.techwell.com

The CMMI Conference: SEPG North America 2014
6-7 May 2014
Washington, DC
http://cmmiinstitute.com/thecmmiconference2014

2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines
11-13 May 2014
Boston, MA
http://www.fccm.org

IBM Edge 2014
19-23 May 2014
The Venetian, Las Vegas
http://www-03.ibm.com/systems/edge

Better Software Conference West
June 1-6, 2014
Las Vegas, NV
http://bscwest.techwell.com

Summer 2014 Software & Supply Chain Assurance (SSCA) Working Group Sessions
24-26 June 2014
McLean, VA
https://buildsecurityin.us-cert.gov/swa

Federated Events on Component-Based Software Engineering and Software Architecture
30 Jun to 04 Jul 2014
Marcq-en-Bareul, France
http://comparch2014.eu

The 26th International Conference on Software Engineering and Knowledge Engineering
1-3 July 2014
Hyatt Regency, Vancouver, Canada
http://www.ksi.edu/seke/seke14.html

SUBSCRIBE TODAY!

To subscribe to CrossTalk, visit www.crosstalkonline.org and click on the subscribe button.

Laws of Immutable Software

Yes, yes, I know. The theme for this issue is “Immutable Laws of Software Development.” Not quite the same as my title. But then, I was struggling for the right topic to write about, and a black lab and a breadbox intervened.

See, we have Molly, a slightly brain-damaged black lab, which my wife rescued back in 2007 before she met me (or, as she oddly refers to it, “the good old days”). Molly had distemper as a puppy. The SPCA wanted to put her down. My wife, having already fallen in love with her, was against the idea. Against all odds (and with much help from the vet), Molly survived, with few physical side effects. Mentally, her brain is permanently stuck in puppy mode. Molly feels that anything on the floor is legally hers, and that anything within reach on the kitchen counter counts as “on the floor.” Leave a loaf of bread sitting out, and it disappears with amazing rapidity. To keep Molly at bay, my wife and I decided to order two rather large breadboxes. When the breadboxes arrived, one was broken.

I called the company and spoke to a very nice customer representative, who quickly apologized, ordered us a replacement, and simply asked us to carefully discard the damaged item—no need to return it. I was chatting with the customer service representative while she was completing the process. She apologized twice to me for the amount of time it was taking, and mentioned, “you have no idea how old this computer system is!” I laughed, told her what I did for a living, and laughingly said, “Are you still running Windows XP?” She laughed back, and replied, “Would you believe MS-DOS?”

I thought she was kidding. Nope. They boot Windows XP, which runs a driver that apparently maps a database of several FAT32 file systems into a set of virtual FAT6 files, and then use command.com to open a MS-DOS window and run a batch file to load a program that was written back in the early 1990s. And, to quote the customer service representative, “it works just fine. It meets our needs.”

In February 2004, my friend and colleague Theron Leishman and I published a Backtalk column entitled “Laws of Software Motion.” [1] In that column, we discovered several laws of software development by comparing them to Newton’s “Laws of Motion.” Newton’s first law is that “An object in uniform non-accelerated motion (or at rest) will remain in the same state of motion unless an outside force acts upon it.” We countered with Cook-Leishman’s First Law – “Any software intensive program not given adequate force (motivation) will degrade and cease to progress.”

Here is a very successful high-end cooking equipment company, with a presence both physical (world-wide) and online. They are using software that was custom-written for them over 20 years ago – and it still meets their needs! Why change to Java from gosh-knows-what? If your company’s software meets your needs, and the cost of keeping it running “as it is” is less than the cost of redevelopment, well then, keep on truckin’. It is called “making a profit”! Sure, you might have to write “glue code” to keep the software working on modern hardware through the years, but it is cheaper than rewriting all the software!

To make software work for 20+ years, you have to do a lot of adaptive maintenance. In the DoD we have quite a few legacy systems that are well over 20 years old. Many of them are interactive, real-time, database-oriented, and interface with customers. If you have never taken on the task of legacy systems maintenance, it’s a different world. It takes a lot (and I mean a lot) of adaptive maintenance to keep them going.

But somehow, we manage to create immutable systems in spite of the “immutable laws.” B-52s still fly (and have been for 60 years). 1960s 70s, and 80s large-scale legacy systems still function. The hardware ages, the hardware gets replaced. Peripherals become obsolete, replace them with new peripherals. We have gone from tapes to floppies (8”, 5 1/4”, and 3 ½”), USBs, CDx, DVD, and now the cloud. And yet, the systems still work.

There is little thrill in working as hard as you can just to keep the system running, pretty much like it was running yesterday, and the month before, and the year before. It is like Alice and the Red Queen in Through the Looking Glass: “Well, in our country,” said Alice, still panting a little, “you would generally get to somewhere else—if you run very fast for a long time, as we have been doing.” “A slow sort of country!” said the Queen. “Now, here, you see, it takes all the running you can do, to keep in the same place.”

It is relatively easy to graduate with a degree in engineering or computer science and develop applications in Java, Objective C, C#, Ruby, .Net, Pearl, or Python. Try becoming fluent in languages of yesteryear, and then transitioning to development frameworks and mindsets from over a quarter of a century ago. Having your college friends laugh at you when you tell them you still program in Jovial, Fortran, or Cobol.

Maintenance programmers, it is your turn. We do not appreciate you enough. Take a bow.

David A. Cook, Ph.D.
(and former maintenance programmer)
Stephen F. Austin State University
cookda@sfasu.edu

(Reference)

1. http://www.crosstalkonline.org/storage/issue-archives/2004/200402/200402-Cook-2.pdf.

Every author yearns to reference himself or herself sooner or later. I feel MUCH better now.

Work for Naval Air Systems Command (NAVAIR) and you'll support our Sailors and Marines by delivering the technologies they need to complete their mission and return home safely. NAVAIR procures, develops, tests and supports Naval aircraft, weapons, and related systems. It's a brain trust comprised of scientists, engineers and business professionals working on the cutting edge of technology.

You don't have to join the military to protect our nation. Become a vital part of NAVAIR, and you'll have a career with endless opportunities. As a civilian employee you'll enjoy more freedom than you thought possible.

Discover more about NAVAIR. Go to **www.navair.navy.mil**.

Equal Opportunity Employer | U.S. Citizenship Required

NAVAIR CIVILIAN

CHOICE IS YOURS.

CROSSTALK thanks the above organizations for providing their support.